(2)

OVERVIEW OF MEGAPROGRAMMING COURSE: LECTURES AND EXERCISES

SPC-93028-CMC

VERSION 02.00.03

FEBRUARY 1994

OVERVIEW OF MEGAPROGRAMMING COURSE: LECTURES AND EXERCISES

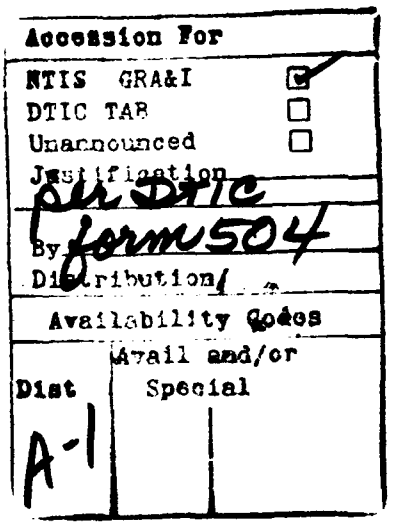
SPC-93028-CMC

VERSION 02.00.03

FEBRUARY 1994

Produced by the
SOFTWARE PRODUCTIVITY CONSORTIUM SERVICES CORPORATION
under contract to the
VIRGINIA CENTER OF EXCELLENCE
FOR SOFTWARE REUSE AND TECHNOLOGY TRANSFER

SPC Building
2214 Rock Hill Road
Herndon, Virginia 22070

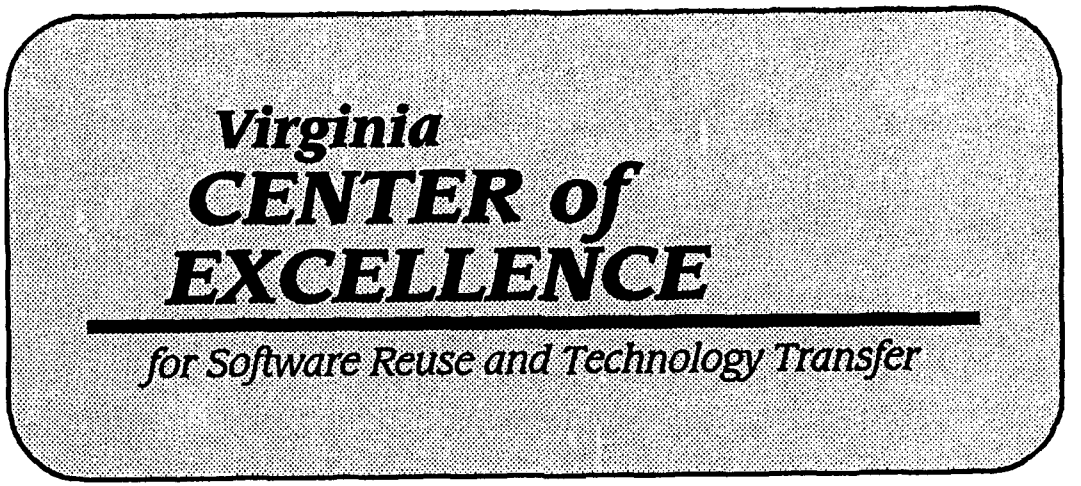


HyperCard is a trademark of Apple Computer, Inc.

Visual Basic is a trademark of Microsoft Corporation.

CONTENTS

This page intentionally left blank.



Overview of Megaprogramming Course

January 1994

This material is based in part upon work sponsored by the Advanced Research Projects Agency under Grant # MDA972-92-J-1018. The content does not necessarily reflect the position or the policy of the U.S. Government, and no official endorsement should be inferred.

To set the stage for the course, ask your students to write down their answers to the following questions:

- How do you write a software program now?
- How do you think you **should** write a software program?

This course will teach your students a better way to perform software development.

****

DISCUSSION

You are learning key computer science principles as you learn how to write software programs in Pascal.

Software development involves more than just writing the code.

The traditional approach to software development has a lot of limitations and problems associated with it.

There's a technique that is different from the traditional software development process.
It's called megaprogramming.

- For the teacher: Megaprogramming is in the research stage, although individual ideas within megaprogramming have been in use within industry for years. Because megaprogramming is in its infancy, some of the terms and some of the specific details may change. The underlying concepts and principles behind megaprogramming are unlikely to change.

Today, we want to explain traditional software development concepts and introduce megaprogramming.

First, we want you to understand why you should care about megaprogramming.

OBJECTIVES FOR THE ENTIRE UNIT

The students should be able to:

- Explain the motivation for megaprogramming
- State that megaprogramming looks at similar problems and solutions together, as opposed to seeing each problem/solution combination as unique

Unit 1: Software Development

DISCUSSION

Computer programming started with machine language, then assembly language, then simple third generation languages (3 GLs—BASIC, FORTRAN, and COBOL), then more complex 3 GLs (Pascal, C, and Ada), then fourth generation languages (spreadsheets, HyperCard, Visual Basic). Megaprogramming is another big step: this may be how most software is developed in the future.

Megaprogramming sees generating software as a process of defining and solving problems, rather than just a programming process. This concept is more in line with what you will see if you get a job involving problem solving with computers.

Because megaprogramrning deals with more than just programming, you first need to understand the "big picture" of how software is typically developed.

STUDENT INTERACTIONS

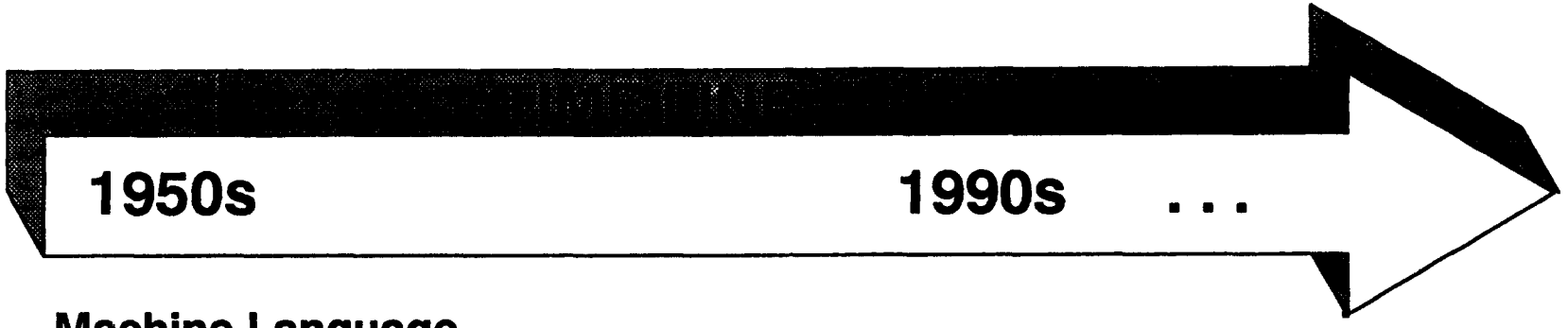
- How many of you have ever programmed in a language other than what you are using in this class? Was it better or worse? Why? These answers should support how each subsequent generation has been better than previous generations.
- Does anybody know what machine language is? Assembly language? Have you written any programs using either? How does it compare to the programming language you have used in this course?
- In movies and TV shows that take place in the future (e.g., Star Trek), what did the computers do? Do you think we could program computers to work like that by programming the way we do now?
- Have you ever thought that developing software would be easier if you possessed some missing knowledge or capability? What was it? Have you ever wished for some software to help you develop software? What type of software would it be?

OBJECTIVE

The students should be able to:

- Understand that megaprogramming is something about which they should be concerned

Evolution of Software Development



- Machine Language
  - Assembly Language
    - Early 3GLs (FORTRAN, BASIC)
      - Advanced 3GLs (Ada, C, Pascal)
        - 4GLs (Spreadsheets)
          - MEGAPROGRAMMING

Virginia CENTER of EXCELLENCE for Software Reuse and Technology Transfer

DISCUSSION

Software is built and used following what industry terms a *life cycle*. A software life cycle describes the key steps in developing and supporting software.

These steps usually include the steps shown on this picture.

- Statement of Need. Your customer asks you to develop a software system that solves some problem.
- Software Development Process. You develop the software system that solves the problem.
- Operation and Maintenance. Your customer uses the software system. You fix any bugs that are found; you add, modify, and delete features as the customer's needs change.
- Retirement. Your customer stops using the software because (1) the software no longer supports a need or (2) a new version of the software is developed that replaces what the customer has been using or makes it obsolete. The software is therefore not used any more.

We are focusing on the second step, the Software Development Process, in this course.

How has software traditionally been developed?

STUDENT INTERACTIONS

- Do you think that the life cycle for software is the same for other nonsoftware products (e.g., automobiles)? Why?
- Can you name a software product that has been retired? Why was it retired? Examples of retired software products are old MS-DOS versions and software that ran on obsolete computers.

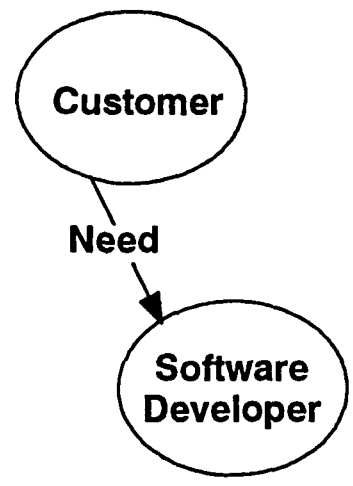
OBJECTIVES

The students should be able to:

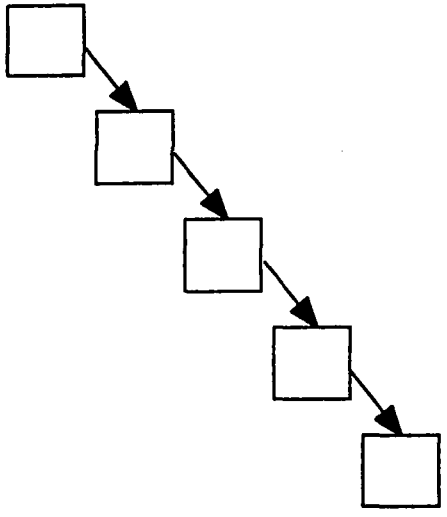
- State that the software developer is always meeting a customer's needs by writing software (currently, the teacher is the customer)
- State that the customer must be supported during use of the software (that is, the job doesn't end once the software is written)

The Big Picture of Software Development

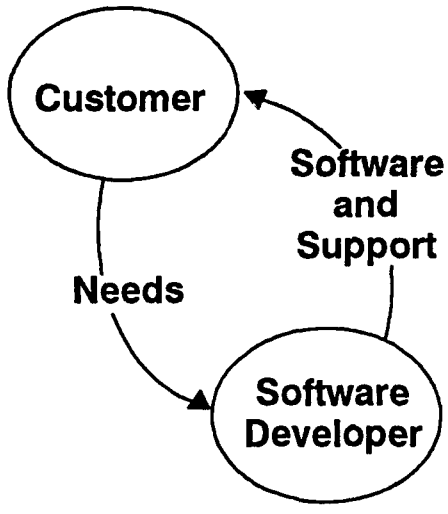
Software Life Cycle



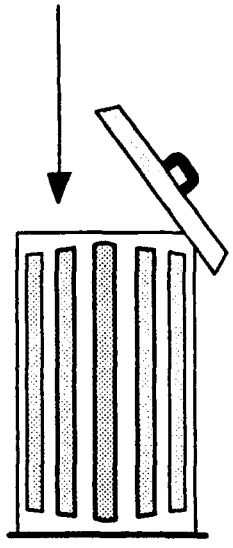
Statement of Need



Software Development Process



Operation and Maintenance



Retirement

DISCUSSION

This is the way software is traditionally developed.

This relates to the programming you've been doing in class:

- Your teacher gives you problems to understand and solve: these are the requirements. In industry, engineers take the customer's general statement of need and translate it into more specific requirements.
- You figure out what procedures you need and the calling hierarchy among them: this is the design.
- You write the code and the documentation.
- You make sure your program solves the problem your teacher gave you: this is testing.
- You turn your homework in: this is delivering the software.

In industry, software is usually developed by teams of engineers. That is, different parts of the entire software system are developed by different teams. During the design phase, the engineers managing the development need to plan carefully how the system is parceled into these different parts (these parts include software and documentation). The engineers then need to make sure that: (1) all of the parts developed by the different teams fit together into one working system; (2) the entire system is tested; and (3) the system is delivered to the customer and supported during use.

Now that you understand some of the "big picture," we can give you some more motivation as to why megaprogramming is needed in the computer industry.

STUDENT INTERACTIONS

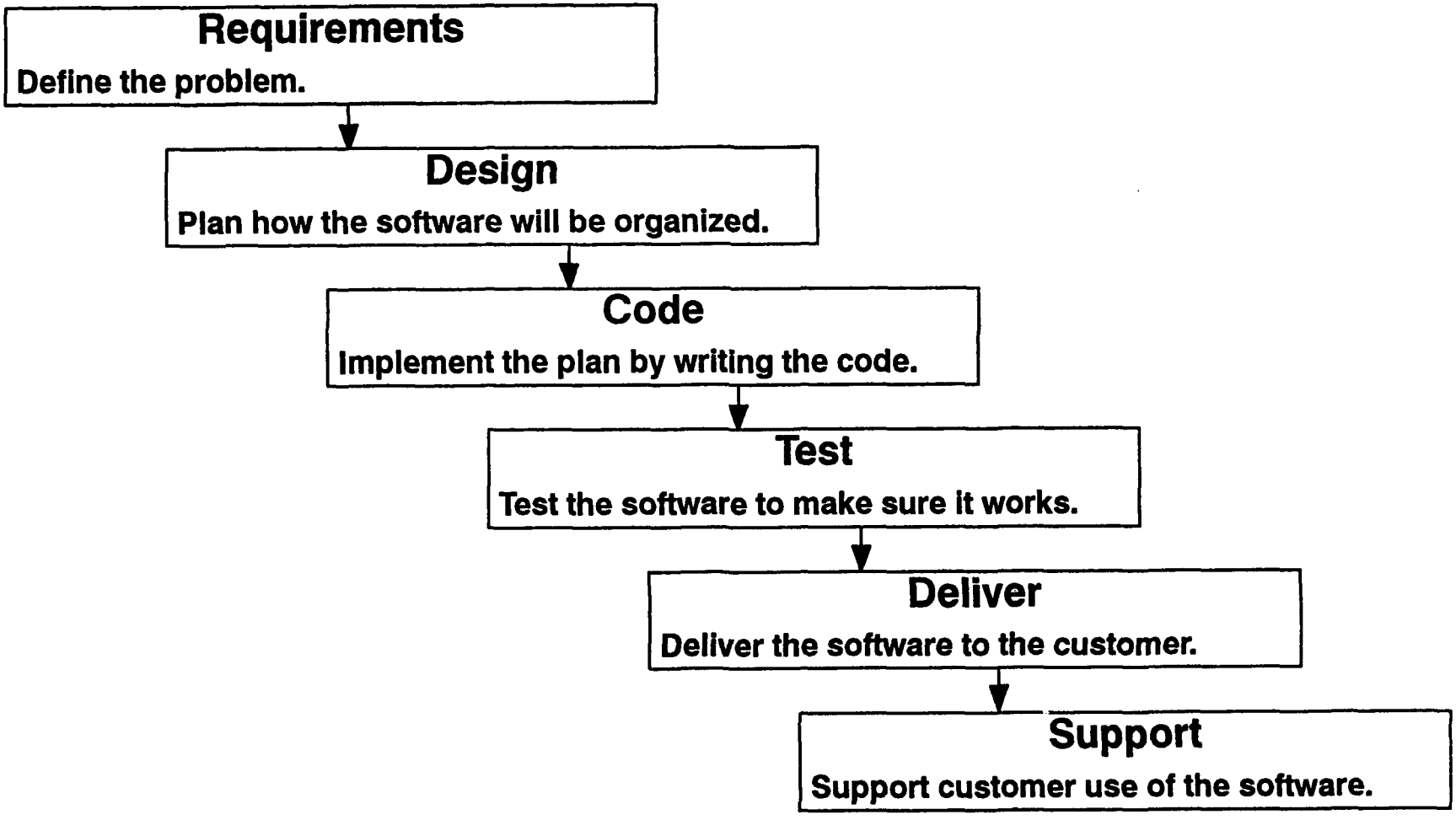
- Must the steps be done in this order? Can you omit one and still develop good software?

OBJECTIVE

The students should be able to:

- Define the key steps in a traditional software development process

The Traditional Software Development Process



DISCUSSION

The chart on the left shows the impact if you don't catch a mistake until later in the software development process: it's four times more expensive to fix it in testing and 100 times more expensive to fix it in maintenance than it is to fix it in the requirements step. This is for the SAME mistake.

Obviously, emphasis on doing things right belongs in the early steps of the software development process (i.e., the requirements and design steps).

In fact, projects usually spend more time in requirements and design than they do in coding. The chart on the right shows percentage of time spent on each of the steps for a typical project in industry. Notice that on a typical project you will only spend 20% of the time on coding. Notice that twice as much time is spent both in requirements and design and in testing than on doing the actual coding.

Together, these charts show that you should, and do, spend a lot of time defining the requirements. In fact, software requirements are the most important step in the process. As we'll see, megaprogramming emphasizes getting the requirements right at the start. This is a valuable contribution. In the traditional software development process, customers often don't discover problems in their requirements until they are using the software. In other words, they have the right solution to the wrong problem. This is no better than having the wrong solution to the right problem!

Why are software requirements so important?

STUDENT INTERACTIONS

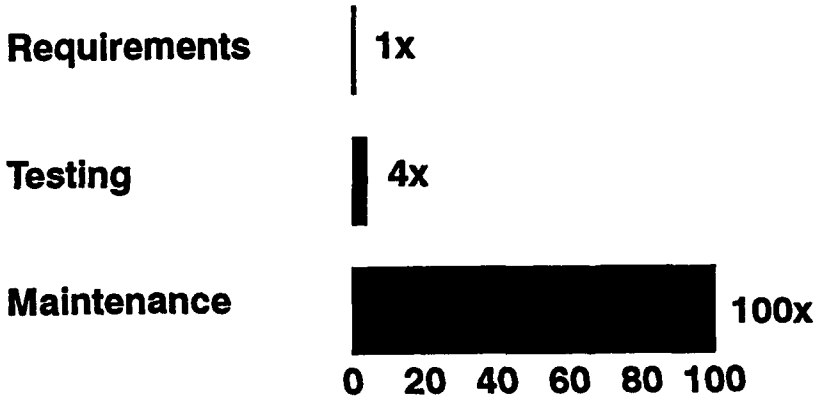
- How much time do you spend in design versus testing versus coding?
- Why do you think it costs so much more to fix a bug in maintenance than it does in requirements?

OBJECTIVES

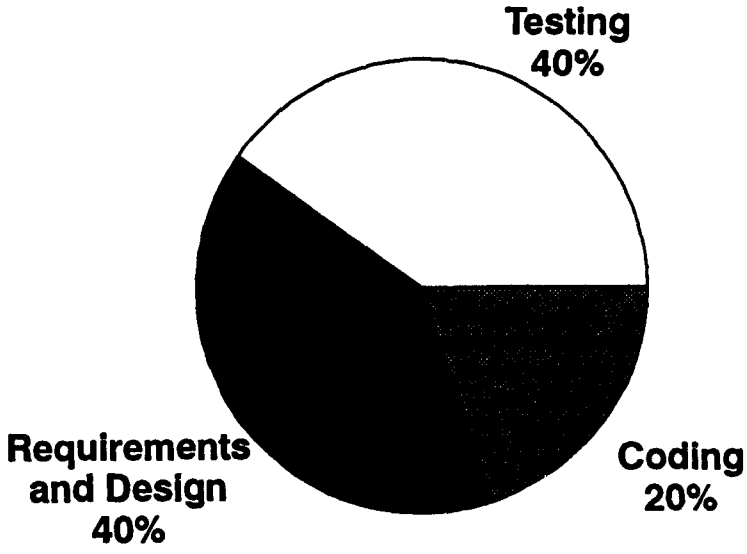
The students should be able to:

- State that writing the code takes up less time than the other steps
- Explain why it is necessary to spend more time in the requirements step

Some Data on Software Development



Relative Cost to Fix the SAME Mistake in Different Software Development Steps



Percentage of Time Spent in Different Software Development Steps

DISCUSSION

Requirements define the problem: they tell you what the software needs to do. The rest of the steps in the traditional software development process create the solution. To create requirements, an engineer takes the customer's general statement of need and turns it into a specific description of what the software should do.

If you don't know exactly what the software is supposed to do (that is, if the requirements are wrong or vague), then how can you do the right thing?

Take the examples on this slide. The bad (vague and untestable) requirements can be interpreted differently by different people (and usually are). What, exactly, does "responsive" mean? What the developer thinks is "responsive" is probably not what the customer thinks is responsive.

Wrong or vague requirements are responsible for some of the biggest problems industry has with the traditional process (in fact, problems in requirements are often cited as industry's #1 software problem):

- Customers often don't know what they really need.
- Customers' actual needs change (even as the software is being developed).
- Customers do not communicate their needs to software developers clearly and completely.

Requirements are hard to write down: they need to be complete, cover all of the details, and work together. This becomes nearly impossible when the number of detailed requirements gets up into the **thousands**. The exercise after this unit will help you understand how hard it is to write detailed, specific requirements.

Megaprogramming helps you in the requirements stage. How does megaprogramming do this?

STUDENT INTERACTIONS

- What have been your requirements for the programs you've been writing in this class? How important have they been to you in writing the right program?
- Which of the requirements in the right column are still vague? How can they be improved?

OBJECTIVE

The students should be able to:

- Explain why the requirements step is the most important step of the software development process

Software Requirements

Examples of Bad (Vague and Untestable) Requirements	Examples of Good (at Least Better) Requirements
The system shall be responsive.	The system shall provide the correct response to 99.9% of user inputs. The system shall provide a response within 1 second to 95% of user inputs.
The system shall be user-friendly.	The system user interface shall employ a graphical user interface and permit input via a mouse.
The system shall be reliable.	The system shall run uninterrupted for at least 1,000 hours during normal use.

Are any of these requirements still vague?

The best design and the best coding will not help if the requirements are wrong.

Virginia CENTER of EXCELLENCE for Software Reuse and Technology Transfer

DISCUSSION

This is a top-level view of the megaprogramming process.

Megaprogramming doesn't look at one problem/solution combination in isolation. Megaprogramming looks at a whole set of problem/solution combinations that are similar in some way. For now, we will call this set a **problem area**.

The purpose of the step above the dotted line is to: (1) define the problem area; and (2) create software and documentation that can be used to create a solution for any particular problem in that problem area.

The purpose of the step below the dotted line is to create a solution for a customer who has a problem in that area. To do this, you use and build on the work done by the engineers who worked to define the problem area.

Everything above the dotted line is done once and then continually improved. Everything below the dotted line is done every time you have to solve a problem in that problem area.

A good analogy to megaprogramming is the building of computers. Most computer companies build several types of computers—for example, desktop and laptop. They don't build all the individual chips and boards differently. They take existing parts and, based on their knowledge of how to build computers, assemble them in different ways depending on what they want to produce. Software developers, on the other hand, traditionally generate software from scratch each time. Megaprogramming is an attempt to allow software developers to do what hardware engineers do: use existing, proven components each time they build a product.

More on this tomorrow (and for the rest of this course).

STUDENT INTERACTIONS

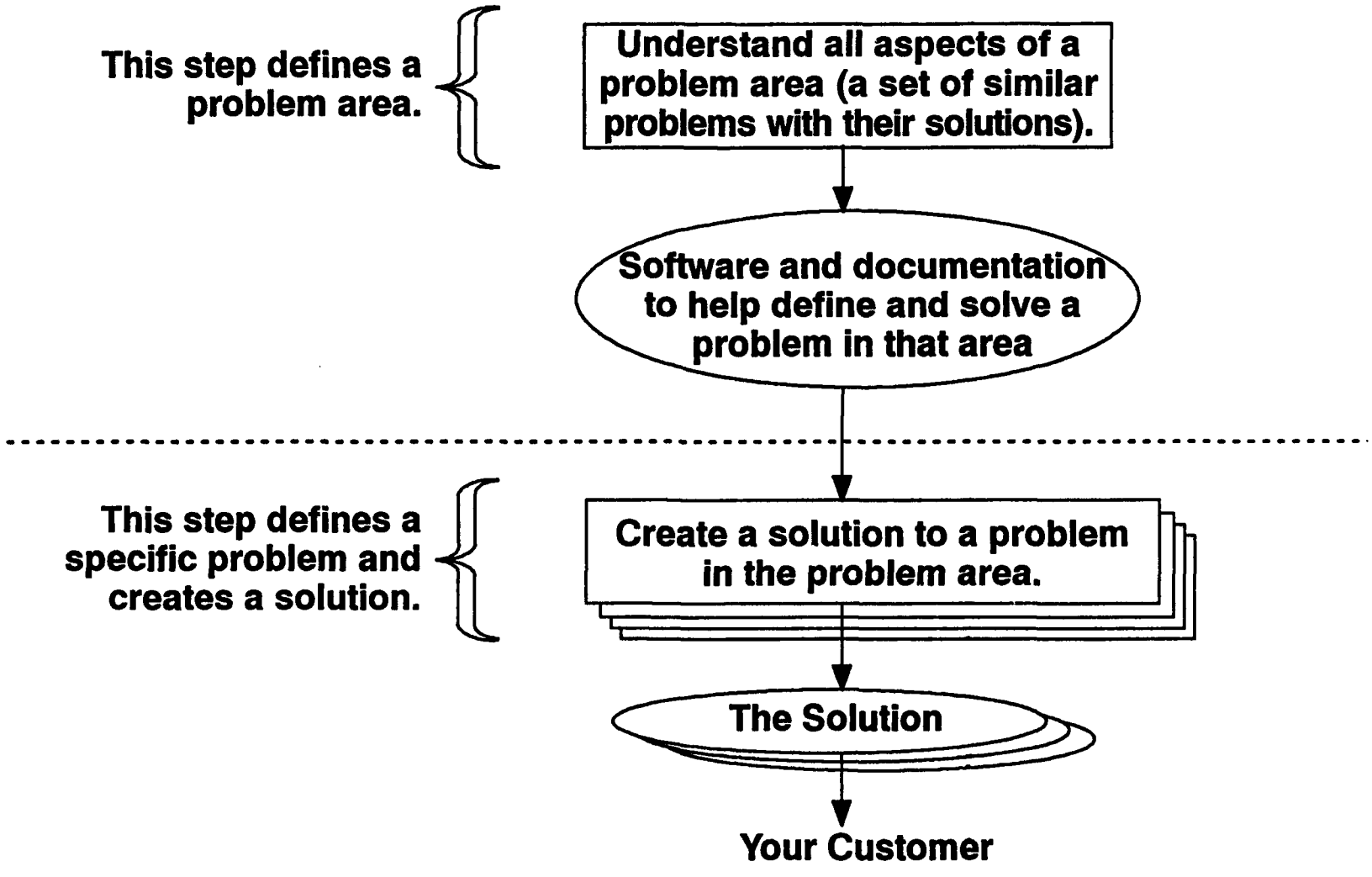
- Does the megaprogramming analogy for computers relate to the building of cars? When the automobile builder builds a car, do they build everything from scratch?

OBJECTIVE

The students should be able to:

- Explain that megaprogramming deals with problem areas (a set of similar problems and solutions)

Megaprogramming



UNIT 1: SOFTWARE DEVELOPMENT

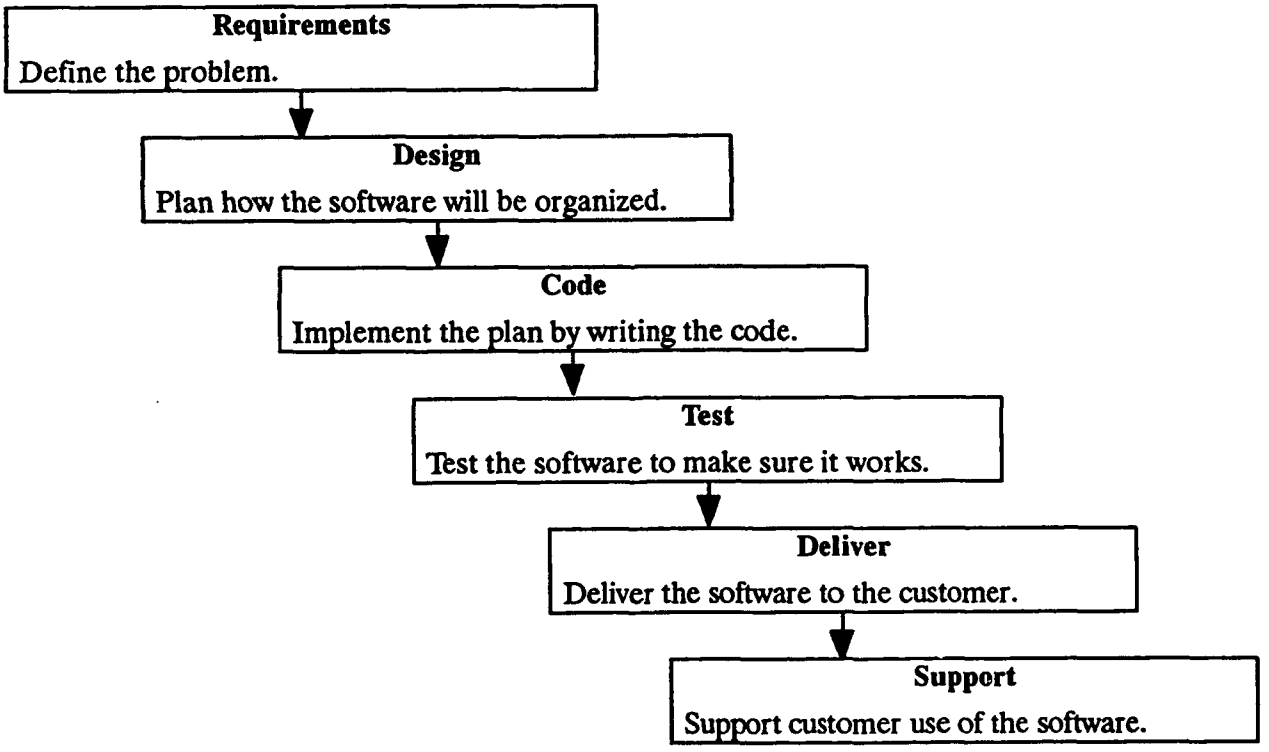
SUMMARY

Software development involves more than just writing code.

SOFTWARE LIFE CYCLE

- The customer states the NEED for the software.
- The developer DEVELOPS the software.
- The software is USED, debugged, and enhanced.
- The software becomes obsolete and is RETIRED.

SOFTWARE DEVELOPMENT PROCESS

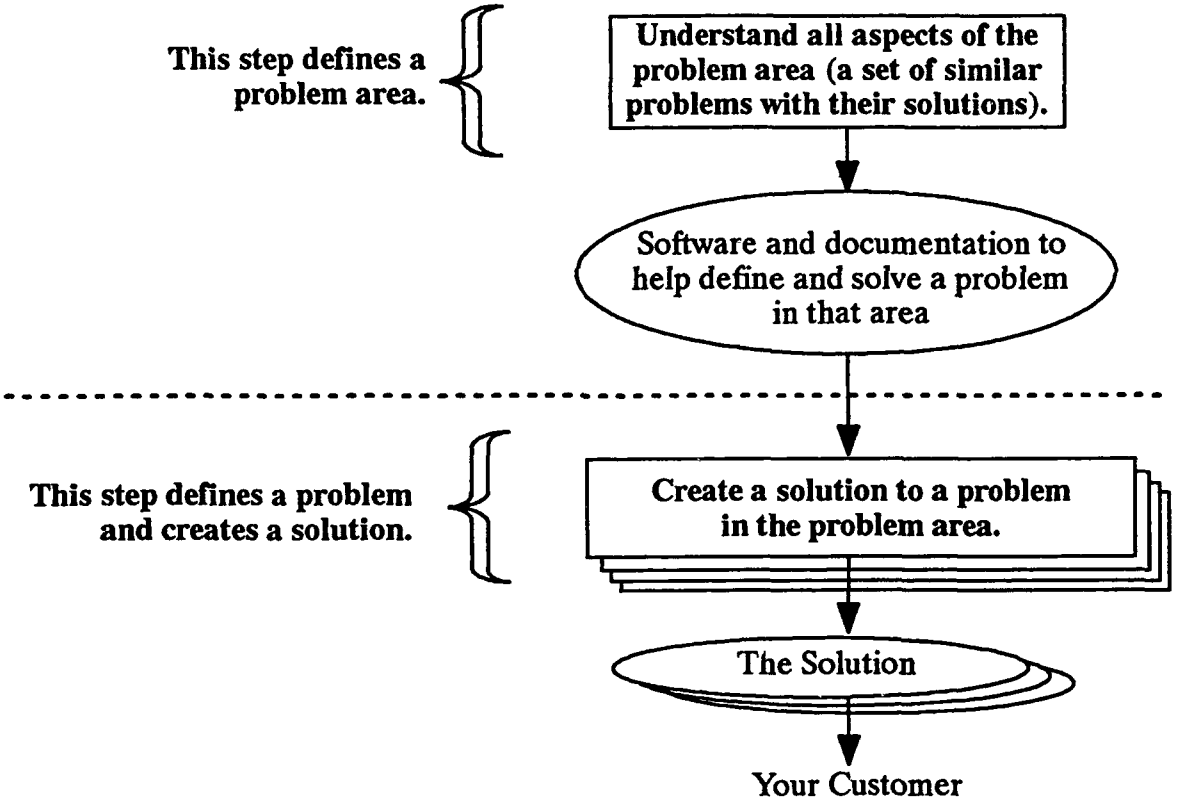


REQUIREMENTS

- Requirements define the problem.
- Since you cannot solve a problem unless you know what the problem is, defining the requirements is the most important step in software development.
- Wrong or vague requirements cost money.

MEGAPROGRAMMING

- Megaprogramming is the next generation in software development processes.
- Megaprogramming looks at similar problems and solutions as opposed to seeing each as unique.
- This set of similar problems with their solutions is called a *problem area.*
- Megaprogramming takes advantage of the similarities and differences between the problems when generating a solution to a specific problem.
- In the following figure, everything above the dotted line defines problems and solutions within the problem area. The steps below the dotted line are done to create a specific solution for a specific problem within that problem area.



A good analogy to megaprogramming is the building of computers. Most computer companies build several types of computers—for example, stationary and laptop. They don't build all the individual chips and boards differently. They take existing parts and, based on their knowledge of how to build computers, assemble them in slightly different ways depending on what they want to produce. Software developers, on the other hand, traditionally generate software from scratch each time. Megaprogramming is an attempt to allow software developers to do what hardware engineers do: use existing, proven components each time a product is created.

UNIT 1: SOFTWARE DEVELOPMENT

EXERCISES

GENERATING REQUIREMENTS

Write down all of the information you think you would need to develop a software program that would solve the following problems. Don't worry about specific procedures. List only all of the information you would need to solve the problem.

1. Beach Trip – You and a friend want to drive to the beach for a weekend and you want to know (1) how long it is going to take and (2) how much the gas is going to cost.

2. Scheduler – You want to develop a program that automatically schedules all of your activities during the week. You want to be able to run this program every Sunday so you know the time, date, and location for each activity.

3. The Cleaning Robot – With such busy lives these days, you decide to develop a robot that will clean up litter in a teenager's room.

4. Several Robots – Suppose you work for United Robot Workers, Inc. (URW). Three customers approach you. Each has different needs:

   a. Customer 1, a farmer, owns a large cornfield and has trouble finding time to harvest it. She wants to know if you can provide a robot that will harvest her corn without human supervision.

   b. Customer 2 is from the Alaska National Guard, which is constantly rescuing people who wander too far afield in the tundra. Mounting a rescue party is time consuming; people have died while the members of the party are gathering. The Guard thinks having robots ready could eliminate these life-threatening delays.

   c. Customer 3, from the National Park Service, is concerned about growing amounts of litter in national parks and wants to know if you can provide a robot that can pick up the litter.

   These three statements correspond to the customers' vague understandings of their problems and of potential solutions. Your task is to write a set of questions for each customer that would clarify each of the problems.

5. Vending machines – The Student Government Association (SGA) has funds to build a vending machine room near the central hall. The principal has agreed to let the SGA go ahead if they make provisions to keep it attractive and litter free. It is your job to define the requirements for the vending machine. What information do you need to define the requirements?

   List the exact requirements for the particular vending machine you were assigned in class. The requirements you come up with will most likely expand beyond the requirements identified in the class discussion.

This page intentionally left blank.

UNIT 1: SOFTWARE DEVELOPMENT

TEACHER NOTES FOR EXERCISES

Here are lists of needed information for each of the problems. Each list is probably not complete. Again, the point of the exercise is not to create a comprehensive list but to make the students realize how hard it is to generate a complete list.

Several of these exercises deal with hardware as opposed to software. However, the course lecture material focuses on software. The point of the exercises for all units is to get across key concepts in software development and in megaprogramming. We have used examples in these exercises that we feel will get the concepts across to the students without worrying about whether or not the example was software-related.

The first three exercises are optional. The fourth exercise begins the introduction of what the students will see in the laboratory. The fifth exercise is threaded into the next day and can be used as homework.

GENERATING REQUIREMENTS

Write down all of the information you think you would need to develop a software program that would solve the following problems. Don't worry about specific procedures. List only all of the information you would need to write the software.

1. Beach Trip – You and a friend want to drive to the beach for a weekend and you want to know (1) how long it is going to take and (2) how much the gas is going to cost.

   - *Cost of the gas*
   - *Number of miles to the beach*
   - *The speed limit*
   - *How much time it takes to stop for gas*
   - *How much gas you have in the tank when you start the trip*
   - *How big your gas tank is*
   - *How many miles per gallon your car takes*

   You also have to make certain assumptions, such as the following (others might make different assumptions, which would change the program and the answer):

   - *You always drive at the speed limit.*
   - *The only time you stop is when you stop to get gas.*
   - *When you stop for gas, you always stop for the same amount of time.*
   - *There is no traffic.*

2. Scheduler – You want to develop a program that automatically schedules all of your activities during the week. You want to be able to run this program every Sunday so you know the time, date, and location for each activity.

   - *A list of all of the activities you are involved in for the week*
     - *Those that are flexible and can be performed on any day (e.g., working on your term paper)*
     - *Those that can only be performed at certain times (e.g., when the computer lab is available for use)*
   - *How long each activity takes*
   - *Whether there are any activities that need to be performed before other activities can start or finish (e.g., you have to practice your piano before your next piano lesson)*
   - *What your start time is for the day*
   - *What your end time is for the day*
   - *If there is a deadline for any of the activities (e.g., your term paper is due on Thursday, so it is better not to schedule that work for Friday)*
   - *How you want your schedule to be presented (e.g., so it looks like a calendar or just a list for each day followed by the time)*

   It might be useful to have a separate text file to hold those activities that occur every week.

3. The Cleaning Robot – With such busy lives these days, you decide to develop a robot that will clean up litter in a teenager's room.

   This one is a lot harder because each teenager has a different room layout and different types of litter.

   - *How often does the room need to be cleaned? This will have an impact on how much litter there is—cleaning once a month means more litter to pick up than cleaning once a week.*
   - *How much litter is in a typical teenager's room when it is time to do the cleaning? This will affect the size of the bag that the robot carries to hold the litter.*
   - *What distinguishes litter from nonlitter?*
   - *What types of litter are there? Is the litter usually small (paper, bottles, cans, wrappers) or might it be bigger?*
   - *Should the robot discriminate among articles it picks up – e.g., clothes on the floor that should go into a laundry basket as opposed to the trash can? Are there other items that should not go into the trash can? What should the robot do with them?*
   - *What does a typical teenager's room look like (for example, what kind of furniture is there)? Do we need to search on top of each piece of furniture for litter or can we look just on the floor?*
   - *How much time does the teenager expect (or can the teenager afford) each cleaning to take? This will have a direct impact on how fast the robot must work.*

Specific information you would need for programming your robot includes:

- *The amount of energy the robot needs (maybe you'll be strapping battery packs on).*
- *The size of the bag your robot will have for litter.*
- *How you plan to make your robot traverse the room. To do this, you will need a map of each room and a strategy for making sure you cover all parts.*
- *How you plan on getting around furniture.*
- *How you plan on sensing the litter (e.g., a metal detector) and the range at which your robot's sensors can sense the litter (e.g., within 1 ft., 3 ft., etc.).*

4. Several Robots – Suppose you work for United Robot Workers, Inc. (URW). Three customers approach you. Each has different needs:

   a. Customer 1, a farmer, owns a large cornfield and has trouble finding time to harvest it. She wants to know if you can provide a robot that will harvest her corn without human supervision.

   b. Customer 2 is from the Alaska National Guard, which is constantly rescuing people who wander too far afield in the tundra. Mounting a rescue party is time consuming; people have died while the members of the party are gathering. The Guard thinks having robots ready could eliminate these life-threatening delays.

   c. Customer 3, from the National Park Service, is concerned about growing amounts of litter in national parks and wants to know if you can provide a robot that can pick up the litter.

   These three statements correspond to the customers' vague understandings of their problems and of potential solutions. Your task is to write a set of questions for each customer to clarify each problem.

   This exercise leads up to the laboratory in Unit 3. There, you will play the role of customer. The scope of the problem area will be restricted considerably more than it is here, making the questions easier to answer. The purpose of this exercise is to get the students thinking about robots. Questions they might pose include, but are not limited to, the following:

   - *How much is the customer willing to spend?*
   - *For the Alaska National Guard, what should the robot do with the people once it finds them? Should it pick them up and carry them to safety, or should it carry shelter and supplies with it? What is an acceptable speed for the robot?*
   - *How much corn (for the cornfield robot) or litter (for the National Park Service) should the robot be able to carry?*

   Remember to make students focus on requirements rather than solutions. They should not ask questions like, "What type of locomotion mechanism do you want?" or "What is the maximum speed of the robot?" As employees of URW, they should already know the answer to such questions.

5. Vending machines – The Student Government Association (SGA) has funds to build a vending machine room near the central hall. The principal nas agreed to let the SGA go ahead if they make provisions to keep it attractive and litter free. It is your job to define the requirements for the vending machine. What information do you need to define the requirements?

The answers to this exercise will follow a different format to support classroom discussion and lay the foundation for a homework exercise and lead-in to a Unit 2 exercise.

It works well to have individuals or groups put their lists on large sheets of paper and tack them to the wall. These lists can then be used in the Unit 2 discussion of similarities and differences.

Class Discussion:

You need to know the following things:

- *What kinds of items will be sold?*

Generate a list of possibilities with the students. The idea here is to have a variety of items. The list might include soft drinks, hot soups, school supplies, snack crackers, fruit juices, nuts and candies, sandwiches, etc.

- *In what price range should the items be?*

(There are a lot more requirements. These are just the first two that will help determine all of the other requirements.)

Generate lists of possibilities for each question. Then, imagine several different vending machines, each fulfilling a different requirements set. Examples:

  - *A sandwich machine that sells only sandwiches and chips. The sandwiches may be hot or cold.*
  - *A soda machine that sells by the can or by the cup.*
  - *A snack machine that sells nuts, crackers, candies, etc.*
  - *A hot meal machine that sells soups, TV dinners, etc.*
  - *A supplies machine that sells pencils, pens, paper, scissors, folders, etc.*

Each one of these vending machines will have its own unique set of requirements. These requirements might include a thermometer to monitor temperature, unique display requirements, varying input support (e.g., bills as well as coins), size of output bin, etc.

Come up with exact requirements for the particular vending machine you were assigned in class. The requirements you come up with will most likely expand beyond the requirements identified in the class discussion.

Assign each student, or small groups of students, one of the machines discussed in class. Their assignment is to list exact requirements for their particular vending machine. The requirements they come up with will most likely expand beyond the requirements identified in the class discussion.

DISCUSSION

In the last unit we talked about software life-cycle processes, traditional software development, and requirements. We also introduced you to the concept of megaprogramming.

In this unit, we will go into more detail on megaprogramming.

OBJECTIVES FOR THE ENTIRE UNIT

The students should be able to:

- Define *domain*
- Determine whether something is or is not a domain
- Define *domain engineering*
- Define *application engineering*
- Explain how domains promote reuse

Unit 2: Concepts of Megaprogramming

DISCUSSION

Fact of life: All people/companies have their own unique problems to solve.

In the example in this slide, two people have two problems to solve. Both are asking for simple math functions, but balancing a checkbook won't help someone learn addition, or vice-versa.

How can they go about solving their problems?

STUDENT INTERACTIONS

- What are the differences between the two problems?
- Suppose you run a business that provides solutions to mathematical problems. What would you do to solve the problems?

OBJECTIVE

The students should be able to:

- Understand that people/customers have unique problems to solve, regardless of how one problem may resemble another

People's Problems

Problem 1

I need something that will help me balance my checkbook.

Problem 2

I need something that will help me teach my little sister how to add and subtract.

DISCUSSION

You can use the following options to solve the problems on the previous slide.

- Option 1. Create two solutions, each one solving its own specific problem. For example, provide a calculator to help balance the checkbook and develop flash cards to help teach basic math skills.
- Option 2. Create a general solution that solves many problems, and try convincing your customers to adopt that solution. For example, provide a computer with multipurpose software that will solve each problem (the spreadsheet can do the math and the Pascal compiler can be used to do the flash cards). This solution, however, may be more than what each of your customers really needs or can afford.
- Option 3. First, understand the problems you are trying to solve. See if both solutions could contain some of the same components. For example, the solutions to both problems require basic math calculations. You could design calculator parts to do standard mathematical functions. You could use these parts to create 2 calculators. One would be a standard calculator that would help the first customer. The other would have an "Ask for answer" button. This would satisfy your second customer.

Doing all the up-front work required in Option 3 to understand the problems pays off in the long run because software developers typically (1) solve many problems during their career and (2) tend to specialize in an area so they end up solving problems that are similar.

How do you determine whether problems have enough in common that you should do the up-front work to see if their solutions could contain some of the same components?

STUDENT INTERACTIONS

- In what field would you like to build software (e.g., to fly space shuttles, to build computer games)? Do you think that there are a lot of commonalities in different software programs developed in that field?

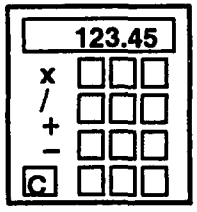
OBJECTIVES

The students should be able to:

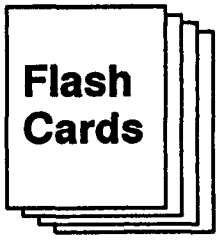
- Explain that there are many ways to solve problems
- Explain that doing a lot of work on understanding the customer's problems pays off in the long run

Solving People's Problems

Option 1: Generate 2 Solutions



and



Option 2: Generate 1 Solution that Solves Both Problems
(*though it is not cost-effective*)

A computer with a spreadsheet and a Pascal compiler

Option 3: Understand How the Problems Are Related Before Generating Any Solution

Problem 1 $\overset{?}{=}$ **Problem 2**

DISCUSSION

To have a cost-effective problem area, you need to make sure that your problems and solutions have enough in common that it makes sense to consider them together. For example, the problem of finding the area of a circle and the problem of solving linear equations do not have a lot in common— it does not pay to consider them together when creating their solutions. It is also not worth the trouble to define your domain such that thousands of problems would fit the definition: this would be too big too manage.

When there is enough in common, we can group these problems/solutions into a problem area instead of looking at each problem and solution as unique. We will now start calling these problem areas *domains*.

Definition: A *domain* is a well-defined area where engineers speak of:

- **Common properties of problems/solutions in the area**
- **Variations among individual problems/solutions in the area**

Engineers have accumulated knowledge and skills: well thought-out laws (e.g., a robot needs to move forward and turn left to traverse any terrain) and engineering judgment (e.g., the amount of energy a robot needs to traverse a forest).

For our lab exercise, we are going to be working with the domain of robots. This domain is based on Karel the Robot, though capabilities have been added to Karel to teach you megaprogramming concepts (in fact, you will create robot programs without writing any software). The domain we have created is based on an imaginary company that builds robots. These robots perform a variety of tasks in a variety of terrains, where the basic goal is to search unknown territory for some type of object. We'll see more of Karel in the lab.

How can domains help us develop software?

STUDENT INTERACTIONS

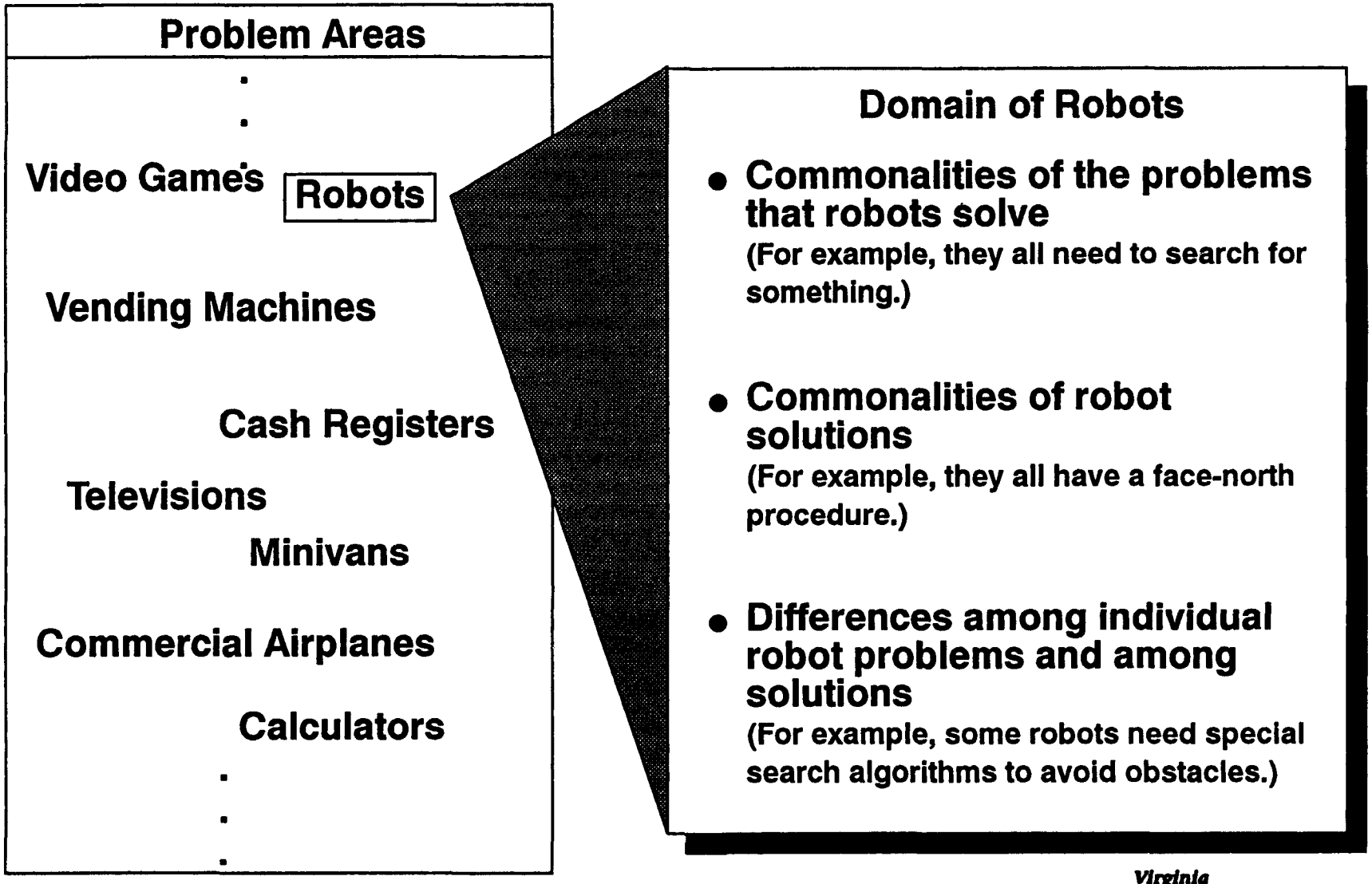
- Have you written procedures that you've used in a lot of your programs? Have there been procedures that would work in only one program?

OBJECTIVES

The students should be able to:

- Understand the concept of a domain
- Be familiar with the robot domain being used for the lab exercise

Domains


Problem Areas
Video Games
Robots
Vending Machines
Cash Registers
Televisions
Minivans
Commercial Airplanes
Calculators
Domain of Robots
• Commonalities of the problems that robots solve
(For example, they all need to search for something.)
• Commonalities of robot solutions
(For example, they all have a face-north procedure.)
• Differences among individual robot problems and among solutions
(For example, some robots need special search algorithms to avoid obstacles.)

DISCUSSION

Domains help us to build software that we can use repeatedly in solving many problems in a domain.

All problems in a domain are variations on a common theme, and we can expect certain similarities among them. These similarities are based on the requirements that are common to each problem in the domain.

All software programs (solutions) in a domain address similar problems. Therefore, these programs should be similar in many ways:

- Many of the procedures that make up each program
- Many of the algorithms used
- Much of the manner in which each program is tested

If we can reuse the procedures that address the similarities for all solutions in the domain, we would have to write these procedures only once.

Whenever we generate a new solution in a domain, then we have to worry about only those parts of the problem that are different from other problems in the domain.

How does megaprogramming incorporate these concepts?

STUDENT INTERACTIONS

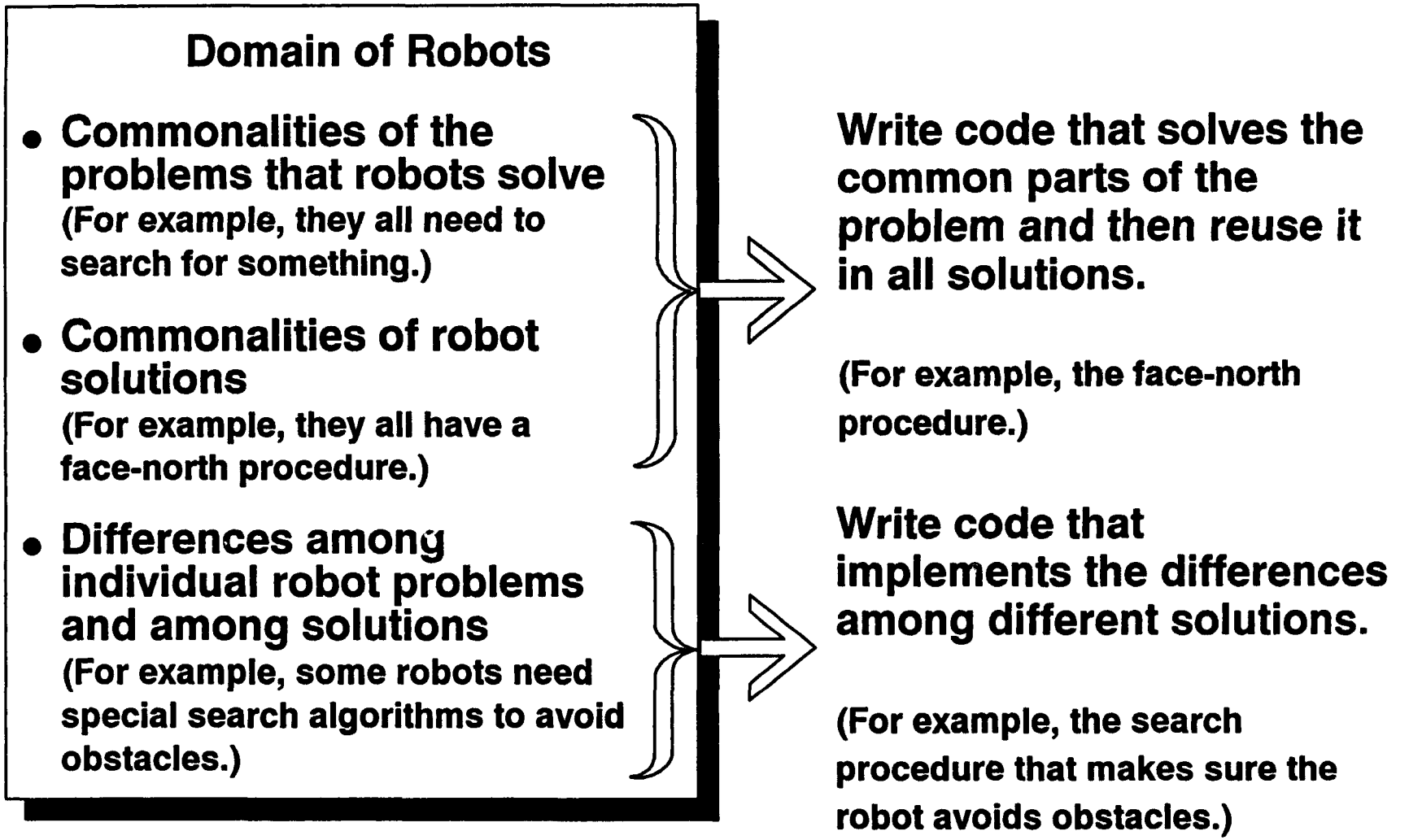
- Can you name other common procedures that you've used in class? What about calls to a COS function? What about calls to a READ operation that reads from a file? Do you consider this reuse? (No)

OBJECTIVES

The students should be able to:

- Use knowledge of a domain to build software procedures that can be used to solve all problems that are in that domain
- Understand how domains support reuse

Reuse


Domain of Robots
• Commonalities of the problems that robots solve
(For example, they all need to search for something.)
• Commonalities of robot solutions
(For example, they all have a face-north procedure.)
• Differences among individual robot problems and among solutions
(For example, some robots need special search algorithms to avoid obstacles.)
Write code that solves the common parts of the problem and then reuse it in all solutions.
(For example, the face-north procedure.)
Write code that implements the differences among different solutions.
(For example, the search procedure that makes sure the robot avoids obstacles.)

DISCUSSION

Now back to our megaprogramming picture.

Let's split software development into two parts:

1. **Domain engineering**, where we:
   - Understand the problems in a domain
   - Determine the best way to create solutions to problems in that domain
   - Create reusable software
2. **Application engineering**, where we:
   - Understand the problem of a particular customer
   - Create a solution to an individual problem of a customer
   - Reuse software we've developed in domain engineering to create our solutions

Tomorrow we'll see what software development is like when we understand a domain

STUDENT INTERACTIONS

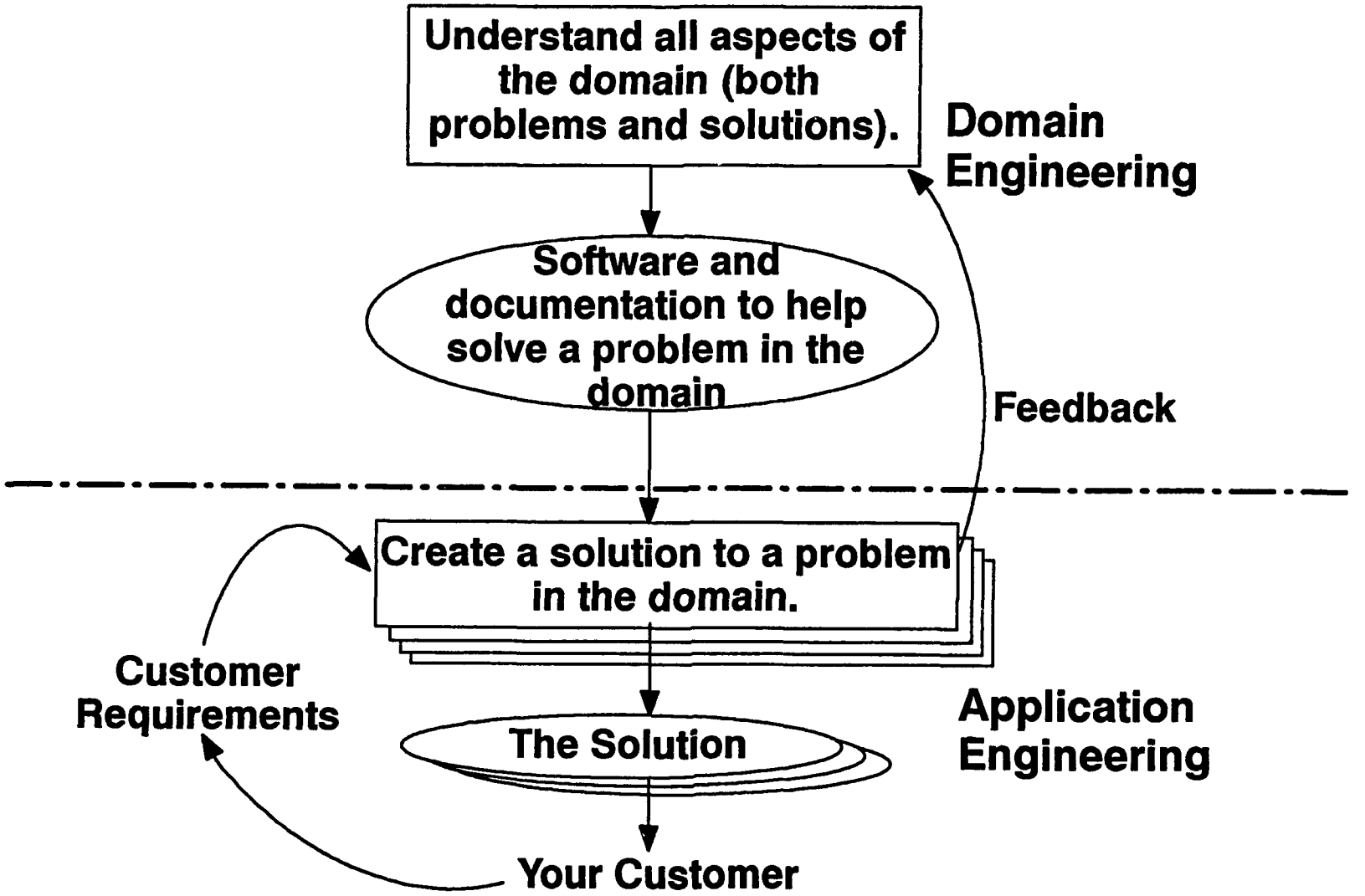
- Would you rather be a domain engineer or an application engineer? Why?

OBJECTIVE

The students should be able to:

- Understand that software development can be split into two parts:
  - Domain engineering, where we understand the problems and prepare solutions for an entire domain
  - Application engineering, where we understand a particular problem and generate a solution for that problem

Megaprogramming

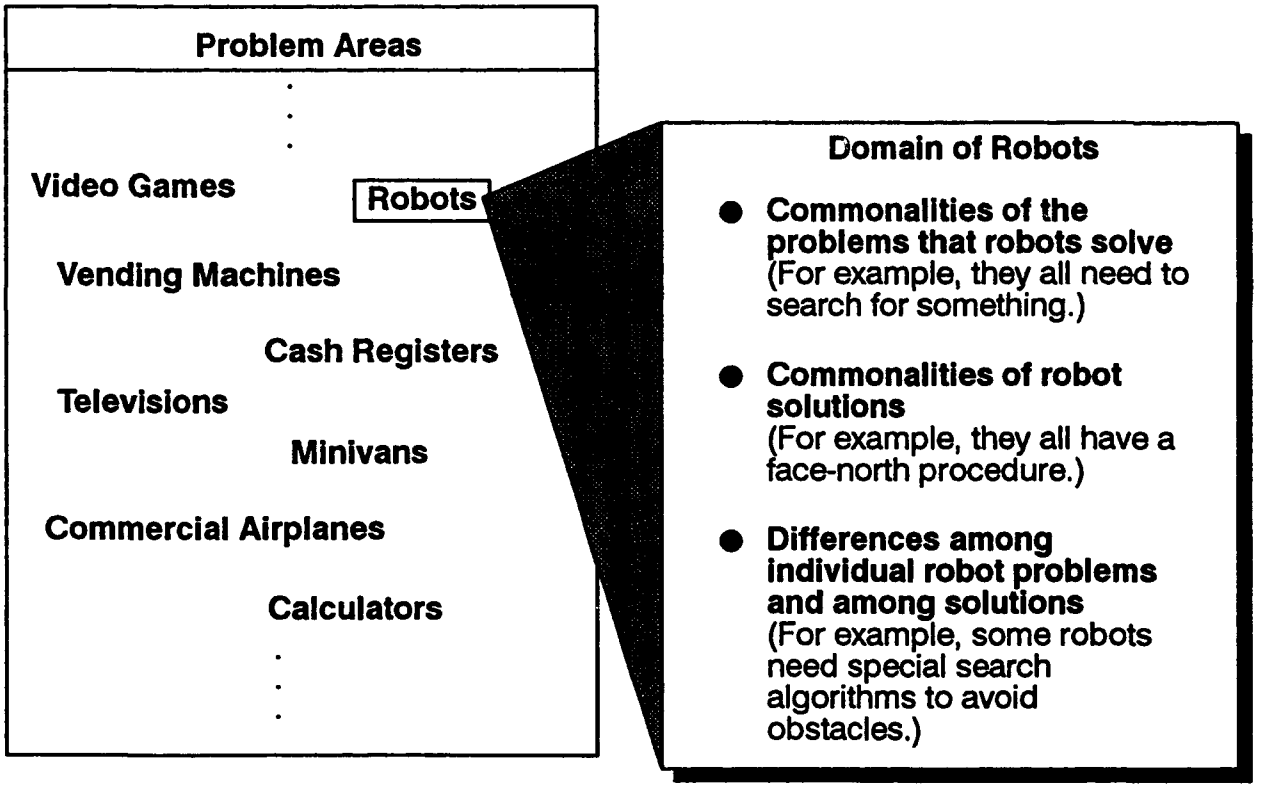


UNIT 2: CONCEPTS OF MEGAPROGRAMMING

SUMMARY

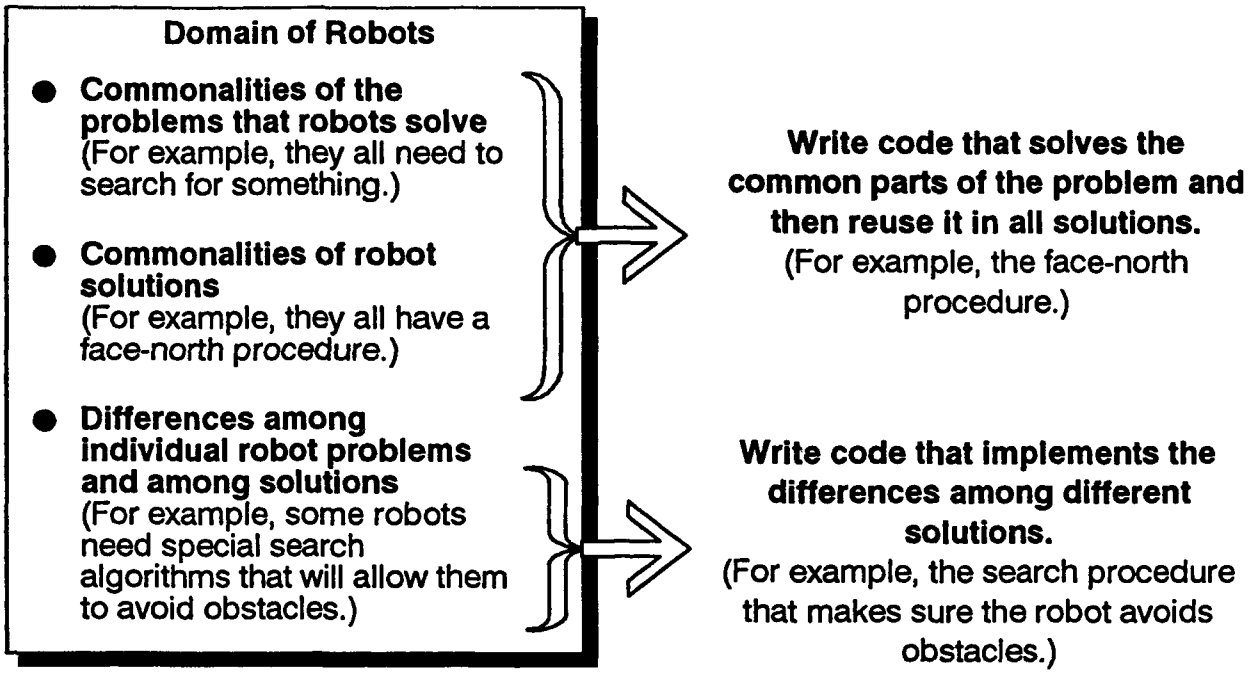
DOMAINS

- Domains contain related problems and solutions that have:
  - Similarities among problems
  - Solutions with common parts
  - Variations among the problems and solutions



- When defining domains:
  - Make sure the problems and solutions have enough in common that it pays to consider them together.
  - Do not include large numbers of barely-related problems in the same domain.
- When identifying a problem in the domain, you only need to identify how it differs from other problems. What is common to all problems defines the other characteristics of the problem.

- When solving a problem in the domain, you can make use of what is common to all solutions.



MEGAPROGRAMMING

Megaprogramming has two main tasks:

1. **Domain engineering**, where we:
   - Understand the problems in a domain
   - Determine the best way to create solutions to problems in that domain
   - Create software that is reusable in all solutions in the domain
2. **Application engineering**, where we:
   - Understand the problem of a particular customer
   - Create a solution to an individual problem of a customer
   - Reuse software we have developed in domain engineering to create our solutions

UNIT 2: CONCEPTS OF MEGAPROGRAMMING

EXERCISES

1. Continue with your vending machine problem (Unit 1, Problem 5). On the board, or on large sheets of paper, list the requirements generated by the students. Ask the following questions:

   Similarities:

   - Are there any similarities among the requirements for the different vending machines?
   - Could a manufacturer design a component for each similarity?

   Differences:

   - What requirements are different from vending machine to vending machine?
   - How could the differences be accommodated? Could any of the differences be a simple modification of an already identified component? Would it be necessary to build an entirely new component?

   Based on these components, what components do you need to come up with for your vending machine? This could include similar components as well as components that are different from all other vending machines. You should also identify which components need to interact with each other, which components you feel are reusable across other vending machines, and which components are unique to your vending machine.

   Each group of students working on a particular vending machine should come up with a list of components that they need to build that vending machine. Each group should present its final list of vending machine components to the class. For each vending machine, discuss the following questions:

   - Have they designed a vending machine?
   - Were they able to identify "reusable" components (i.e., components that could be used with little or no modification)?
   - What components did they have to create to handle requirements unique to their vending machine?
   - Would they consider vending machines a class of common problems and solutions (a domain)?
   - What are some of the benefits of going through these steps?

   When you are finished, answer the following questions:

- Could you use megaprogramming concepts to help build vending machines?
- What would the domain engineer do in this domain?
- What would an application engineer do in this domain?

2. Discuss the robot problem from Unit 1, Problem 4.
   - Make a list of common jobs and tasks that the three robots in Unit 1, Problem 4 needed.
   - Make a list of specific jobs and tasks that not all the robots needed.

HOMEWORK

1. Consider the following—are they domains? Why or why not?
   - The process of applying to college
   - The process of proving equations
   - The process of school bus scheduling
   - The process of transportation scheduling
2. Describe a domain in today's world of teenagers. List the similarities and differences in your domain.

UNIT 2: CONCEPTS OF MEGAPROGRAMMING

TEACHER NOTES FOR EXERCISES

1. Continue with your vending machine problem (Unit 1, Problem 5). On the board, or on large sheets of paper, list the requirements generated by the students. Ask the following questions:

   Similarities:

   - Are there any similarities among the requirements for the different vending machines?

     Examples of similarities might include the need for the following: temperature monitor, display, input, output, storage modules, etc.

   - Could a manufacturer design a component for each similarity?

     This should generate a list such as coin boxes, mechanisms to deliver the merchandise, display units, utilities units, storage units, housing units.

   Differences:

   - What requirements are different from vending machine to vending machine?

     Examples of differences might include a microwave to heat an item, a special option that makes change for bills, etc.

   - How could the differences be accommodated? Could any of the differences be a simple modification of an already identified component? Would it be necessary to build an entirely new component?

     For example: A microwave to heat an item would probably have to be a new component. A bill changer could probably be a modification of the existing coin/bill input mechanism.

   Based on these components, what components do you need to come up with for your vending machine? This could include similar components as well as components that are different from all other vending machines. You should also identify which components need to interact with each other, which components you feel are reusable across other vending machines, and which components are unique to your vending machine.

   Assign a group of students to each of the vending machines identified in the Unit 1 exercise. Based on the components discussed today, have them identify what components they will need to come up with for a complete vending machine. This could include similar components as well as components that are different from all other vending machines. They should also identify which components need to interact with each other, which components they feel are reusable across other vending machines, and which components are unique to this vending machine.

   This can be done either as a homework assignment or as a small-group exercise at the end of Unit 2 or before Unit 3.

Each group of students working on a particular vending machine should come up with a list of components that they need to build that vending machine. Each group should present its final list of vending machine components to the class. For each vending machine, discuss the following questions:

- Have they designed a vending machine?

  See if the other students can identify any missing components. The point of this question is to help the students see that, like requirements, making sure that you have everything is difficult.

- Were you able to identify "reusable" components (i.e., components that could be used with little or no modification)?

  The students should understand why having reusable components can save time and money. These components can be software programs or actual vending machine hardware components: the idea of savings remains the same.

- What components did they have to create to handle requirements unique to their vending machine?

  All solutions will have unique parts. If there were no unique parts, then the solution would be exactly identical to another problem/solution and you would only have to build one solution.

- Would they consider vending machines a class of common problems and solutions (a domain)?

  Yes. There are enough similarities to make it worth your while to understand the similarities and differences among vending machines and to make use of that knowledge each time you build a new one.

- What are some of the benefits of this procedure?

  Savings in design, savings in manufacturing, aesthetic uniformity, etc.

When you are finished, answer the following questions:

- Could you use megaprogramming concepts to help build vending machines?

  Yes. There is enough in common between vending machines, yet enough differences, that it makes sense to study their similarities and differences.

- What would the domain engineer do in this domain?

  The domain engineer would create reusable vending machine components and documents that describe how to use those components to build vending machines.

- What would an application engineer do in this domain?

  An application engineer would talk to a customer and use the products created by the domain engineers to define and validate requirements that met the customer's need, and build a vending machine that satisfied those requirements.

2. Discuss the robot problem from Unit 1, Problem 4.

   - Make a list of common jobs and tasks that the three robots in Unit 1, Problem 4 needed.

   - Make a list of specific jobs and tasks that not all the robots needed.

   The answer to these two questions depends on the students' answers to Problem 4 in Unit 1. However, they might observe that all the robots move, and they search for some type of object. The type of object, and the robot's response to finding it, are two things that vary among the three robots.

TEACHER NOTES FOR HOMEWORK

1. Consider the following—are they domains? Why or why not?

   - The process of applying to college

     Yes. Colleges usually ask for many similar types of information on their application forms, yet there are enough differences that you could not use the same application at more than one school without any changes.

   - The process of proving equations

     Yes. You follow similar steps in solving any equation. However, the order in which you follow the steps and the exact steps you follow will vary from equation to equation.

   - The process of school bus scheduling

     Yes. School bus scheduling will have the same coordination and logistics problems from school to school and county to county. However, there will be enough differences (e.g., number of buses, size of district, etc.) that you could not use the same school bus scheduling system for every school.

   - The process of transportation scheduling

     No. This domain would be too large to justify establishing a domain. There are similarities between different types of transportation; however, there are too many differences from one transportation type to another and not enough similarities that it will not pay to generate and use the domain.

2. Describe a domain in today's world of teenagers. List the similarities and differences in your domain.

   - *The answers for this question will vary. Look for a domain that has enough similarities between the problems and solutions and significant differences that it would make sense to establish and use a domain whenever you need to generate a solution.*

This page intentionally left blank.

DISCUSSION

Unit 2 introduced domains and the importance of considering problems and solutions in the context of a domain.

In Unit 3, we'll see how domains can simplify software development. We'll concentrate on our robot domain and show a straightforward, step-by-step process for developing robot software. In the laboratory, you'll get the chance to use this process.

OBJECTIVES FOR THE ENTIRE UNIT

The students should be able to:

- Understand that a problem can be defined solely in terms of its differences from other problems in its domain
- Understand that an application can be developed by following a well-defined process that involves:
  - Requirements definition
  - Validation
  - Solution generation from precisely-defined requirements

Unit 3: Application Engineering

DISCUSSION

This slide ties concepts from Unit 2 about problems versus solutions to a software development process for defining problems and creating solutions to those problems.

The prototypical steps in developing software—using megaprogramming or otherwise—are as follows. Customers start with a vague understanding of an existing problem and a realization of the need for a solution to that problem (symbolized by the outline of the robot).

- Step 1: The application engineer states a customer's problem precisely (the vision of the complete robot). The ability to state the problem precisely is a measure of how well the problem is understood. The application engineer's knowledge about the problem area helps in reasoning about what robot the customer needs (hence the arrows both from and to the application engineer).
- Step 2: The application engineer generates a solution to the problem based on the problem statement created for the customer (the solution is symbolized by the robot).

This slide does not show what happens after the software is developed, although slides in earlier units did.

Ideally, we as application engineers should always think in terms of problems and solutions as we develop software. We should keep these two parts separate in our minds.

A precise statement of a problem defines the requirements for a solution. As we saw in Unit 1, it's important that we be precise when we define requirements.

What do these two activities (precisely stating the problem and generating a solution) really mean? Let's explore this question, in terms of megaprogramming.

STUDENT INTERACTIONS

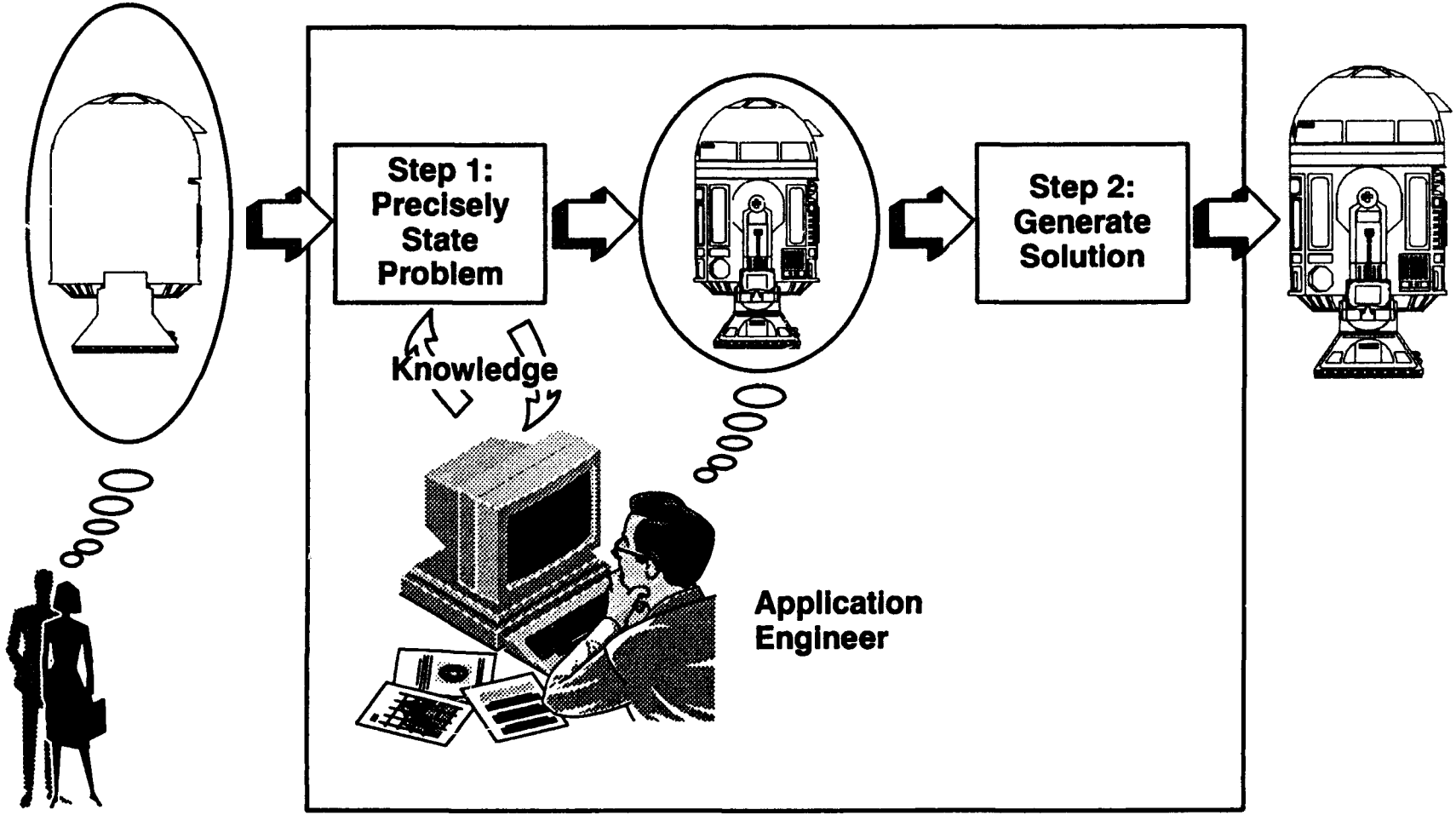
- For the vending machine exercise, did you precisely state the problem? Did you miss any parts of the problem? Could somebody generate a vending machine from your list of requirements?

OBJECTIVES

The students should be able to:

- Explain the notion of software development as a two-step process: precisely stating the problem and generating the solution
- Explain why requirements are the outcome of precisely stating a problem

Application Engineering



DISCUSSION

This slide illustrates how we can precisely state a problem in the context of a domain, once we have analyzed all problems in a domain to the point where we understand the similarities and differences among them.

In general, precisely stating problems so that other people can understand what we mean is hard. We must be unambiguous. We cannot assume people understand anything we do not explicitly state. However, precisely stating a problem is much easier if we know people share certain assumptions about the problem—that is, the domain of which it is a part. These assumptions can be:

- Things we know to be common in all problems in the domain. For example, all people who buy automobiles have a transportation problem that they assume a car will solve.
- Things we know to vary across problems in the domain. Some people who buy automobiles haul cargo. Others haul families. Still others want a sports car. No single car is ideal for all three purposes.

If we really understand a domain, we can state a problem strictly in terms of variations among problems. A decision tree is one way to do this (note that what's on the slide is compressed; it's not actually a tree). For cars, a decision tree can lead us to identify a particular car with specific options. Each tree level represents a statement about your problem that will lead to a different solution. The first level tells you whether you should buy a pick-up truck, a sedan, or a convertible. The second level tells you if you need four-wheel drive. The third tells you if you need air conditioning, etc. (The order among the levels is arbitrary.)

This is pretty much how you decide what car to buy. It works because you and the auto dealer both know what an automobile is. You share a common vocabulary ("coupe" versus "hatchback"). Suppose you didn't share that vocabulary—how would you explain what you wanted? You would have to describe everything (number of wheels, horsepower, number of seats, etc.). But you do share a vocabulary, so you have to only describe how your transportation needs differ from those of other people.

When you buy an automobile, the salesperson acts as an application engineer. (This analogy isn't perfect, because an application engineer typically knows more about the domain and product than the customer.)

STUDENT INTERACTIONS

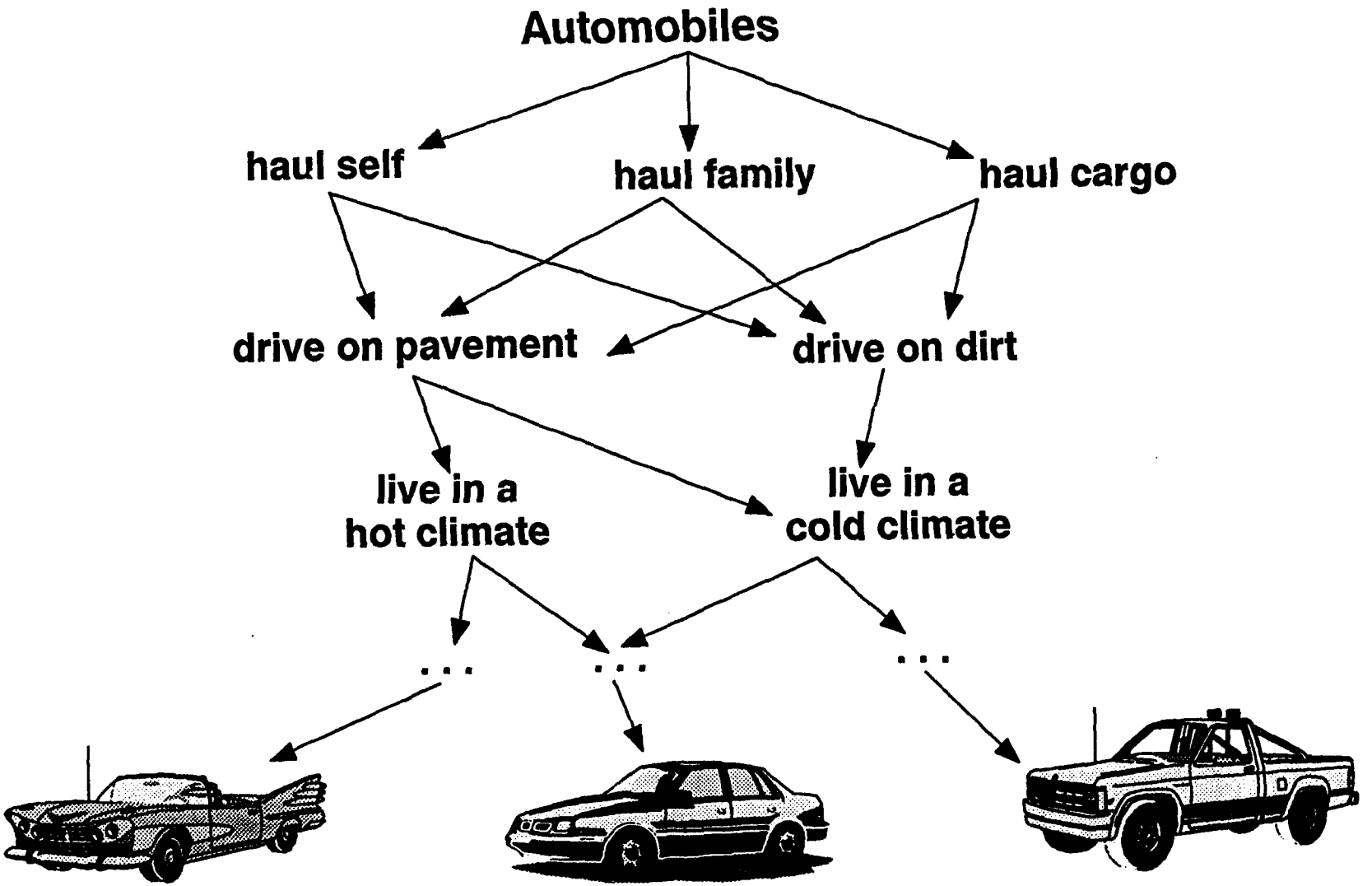
- Name another common situation where you make assumptions about what other people know. (For example, when you purchase a stereo, you assume it will have a volume control.)

OBJECTIVES

The students should be able to:

- Explain that, within a domain, one problem can be defined in terms of its differences from other problems
- Explain a few of the commonalities and variations for the automobile domain

Precisely Stating Problems



DISCUSSION

We can study domains other than automobiles, too. Domains of software can be described in terms of commonalities and differences. Consider our robot domain. The following are examples of characteristics that all robots in the domain share:

- All robots move.
- All robots search for some type of object.
- All robots are autonomous: once started, they perform their mission without human intervention.

Similarly, we can identify how a robot differs from other robots in the domain:

- The type of terrain in which it operates
- The type of object for which it searches
- The type of mission it performs

Unless you understand what's common to all robots in the domain, these differences do not make much sense. But interpreted in the context of things common to all problems in a domain, these differences are enough to precisely state a single problem (and so to define the requirements for a solution to that problem).

(As with Slide 3-3, space limitations necessitated a compressed decision tree. A leaf does not uniquely identify a set of decisions, although following a path from the root to a leaf does.)

Is there a process we can follow to precisely state the problem?

STUDENT INTERACTIONS

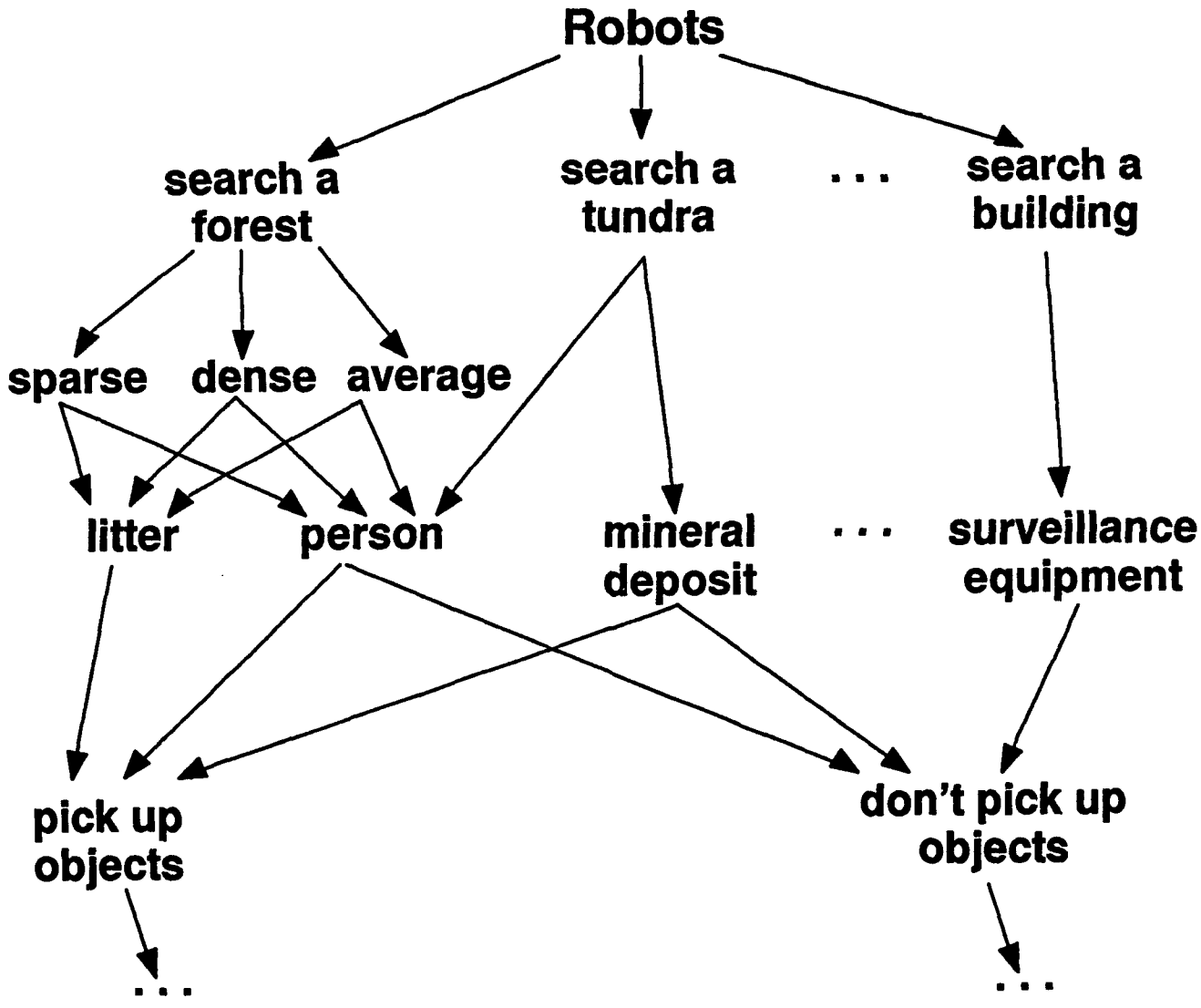
- What would a decision tree look like for buying a stereo?

OBJECTIVES

The students should be able to:

- Explain that a particular problem in a domain can be defined in terms of how it differs from other problems in the domain
- Explain a few of the commonalities and variations for the robot domain

Precisely Stating Problems (cont.)



DISCUSSION

This is a process for creating requirements for robots in our robot domain. Just as with automobiles, we have reduced creating robots to making a finite number of decisions:

1. In what type of terrain will the robot operate?
2. How should it search the terrain?
3. If it's searching a forest, how dense is the forest?
4. For what type of objects will it search?
5. Should it pick up objects as it finds them?
6. Where should it end up?
7. How many objects should it be able to carry?
8. How many batteries will it need?

The application engineer works with customers to resolve these decisions. The result is a complete set of requirements. Note that you can't make sense of a set of decisions without knowing what they mean in the context of a domain. For instance, "forest, sweep,average, litter, yes, its origin, 200, 25" is a valid set of answers for the above questions. But you can't really tell what that means unless you understand the commonalities.

The decisions influence each other:

- Robots that operate in a field don't search for the same types of objects as robots that operate in a tundra.
- Robots whose mission is to search for objects but not pick them up don't need any carrying capacity.
- People familiar with the domain know the best searching strategy (sweep or zigzag), except for robots that operate in a forest.

In other words, the answer given for one decision may affect the range of answers that are valid for subsequent decisions, or even whether the decision must be made. For this reason, the decisions are ordered into a process.

How does the application engineer find out the answers to these decisions? The customer always knows the answers to some (e.g., terrain). The customer may not know the answers to others (e.g., number of batteries). The application engineer must use "engineering judgment." How many objects is the robot expected to carry on an average mission? How heavy are they? This will affect the number of batteries needed.

How do customers know if the requirements really meet their initial vague understanding of the problem?

STUDENT INTERACTIONS

- When buying a stereo, does it matter how the decisions are ordered?

OBJECTIVE

The students should be able to:

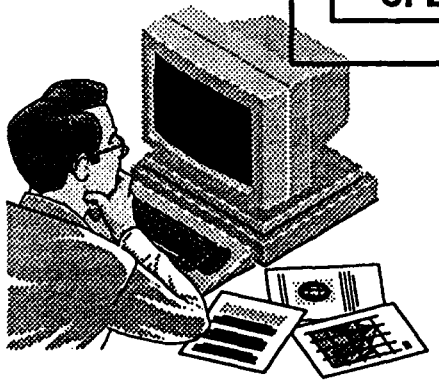
- Explain the process for creating robot requirements

Precisely Stating Problem for a Robot

Customers

Step 1: Precisely State Problem

1. Choose Terrain Type
2. Choose Search Strategy
3. Choose Forest Density
4. Choose Object Type
5. Choose if Objects are to be Carried
6. Choose Ending Position
7. Choose Carrying Capacity
8. Choose Number of Batteries



Application Engineer

Robot Requirements

Step 2: Generate Solution

DISCUSSION

This slide introduces validation into the software process.

Before generating a solution, the application engineer needs to know if the requirements really capture the customer's problem. It's important to do this early, while fixing errors is relatively inexpensive (see Slide 1-6).

We enlarge our view of the process for precisely stating a problem to include validating requirements.

The application engineer makes sure the requirements really express the behavior the customer intended. For instance, does the robot really perform the mission the customer needs? Does the robot have enough batteries to perform its mission in the forest where its customer wants to use it?

As before, the result of the whole "Precisely State Problem" step is the robot requirements. This slide shows that the requirements have an intermediate, nonvalidated form.

Validating requirements is done by a variety of techniques. Simulation and analytical models are often used. For example, application engineers might calculate the number of batteries by using an equation that describes a robot's battery consumption to calculate whether the robot can perform its mission.

If the application engineer's statement of the problem is not valid, the application engineer needs to rethink certain decisions (the arrow leading back to the decision-making portion of the process).

Okay, the customer thinks the requirements are correct. What's the best way for the application engineer to generate a solution?

STUDENT INTERACTIONS

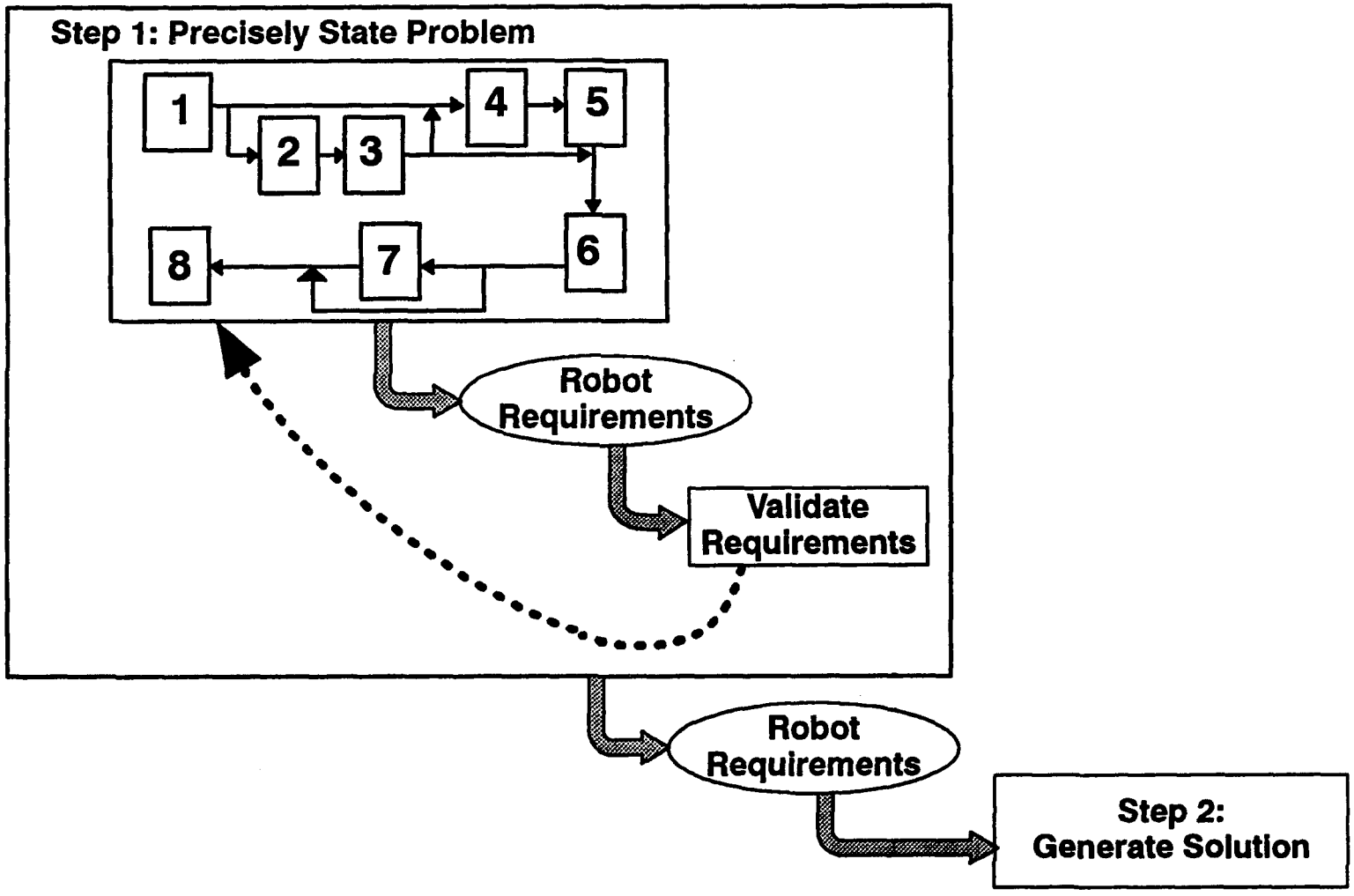
- Can you imagine how a customer paying $10 million for development of a software program feels when, upon receipt, they find that it wasn't really what they wanted? THIS HAPPENS! What would you do? Validating requirements will help that problem.

OBJECTIVE

The students should be able to:

- Understand the role of validating requirements in application engineering

Validating Requirements



DISCUSSION

This slide shows the main concepts in generating solutions from requirements. It also shows how existing software can be reused.

Assume the application engineer's company understands the robot domain, having built some robots for previous customers. That means old parts should be available (from domain engineering) to build robots.

Think of the robot as consisting of two types of parts: (1) parts needed by all robots in the domain plus (2) parts needed by the particular robot the customer needs.

Since parts are available, the application engineer can build a robot by:

1. Selecting the necessary parts (for instance, all robots need an object sensor and a compass; a robot in a tundra always needs a rock-avoiding algorithm, but a robot in a cornfield doesn't).
2. Composing them to create a robot. This works for both hardware parts and software parts!

The requirements are precise enough to identify a particular robot; that means they are also precise enough to identify the set of parts the application engineer needs to build that robot.

Once the application engineer composes those parts, the robot is built.

Ideally, the application engineer won't even need to build any parts—if the right previously built parts are available. (If the customer wants a robot with features that haven't been built before, the application engineer will need to create them.) Also, if someone could figure out in advance which parts are needed to solve which problems, selection and composition could be completely automated. This is possible—it's the role of domain engineering, as we'll see in Unit 4. In the laboratory for Unit 3, we'll see examples of using domain engineering products to build robot software automatically.

STUDENT INTERACTIONS

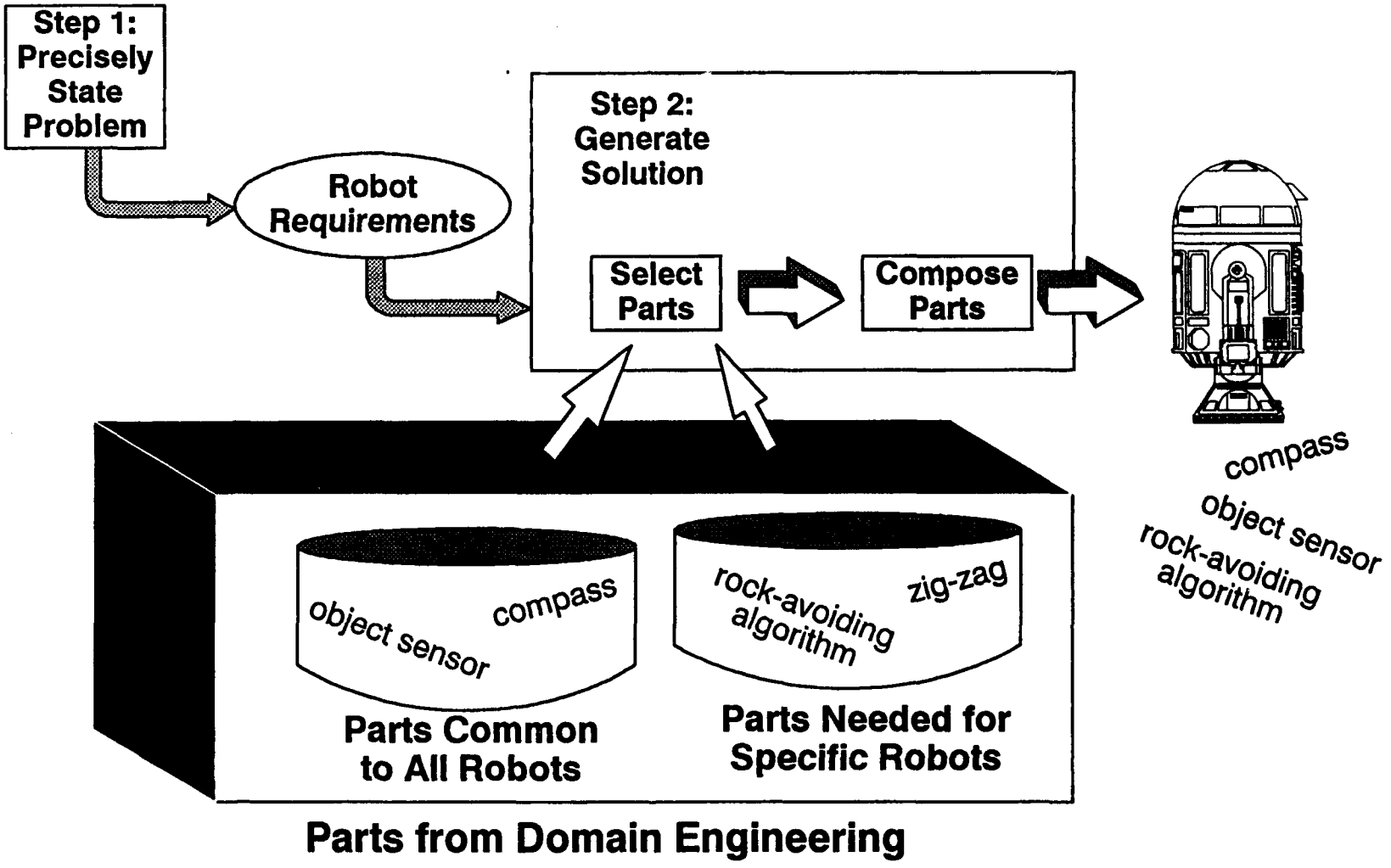
- How would this process work for the vending machine domain? for the automobile domain?

OBJECTIVE

The students should be able to:

- Explain the purpose of the second step of application engineering in developing software: Generating a Solution

Generating a Solution



UNIT 3: APPLICATION ENGINEERING

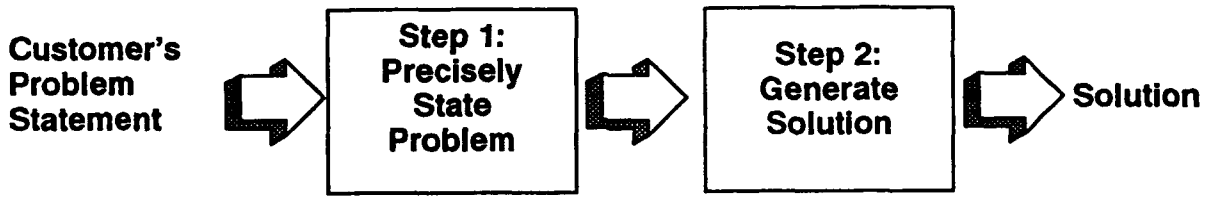
SUMMARY

Application engineering involves:

- A customer who has a problem
- An application engineer who solves the problem

An application engineer solves the problem by:

1. Understanding and precisely stating the problem AND
2. Generating a solution based on the problem statement



STEP 1: PRECISELY STATE PROBLEM

To understand a problem, it is easier if the application engineer understands other related problems:

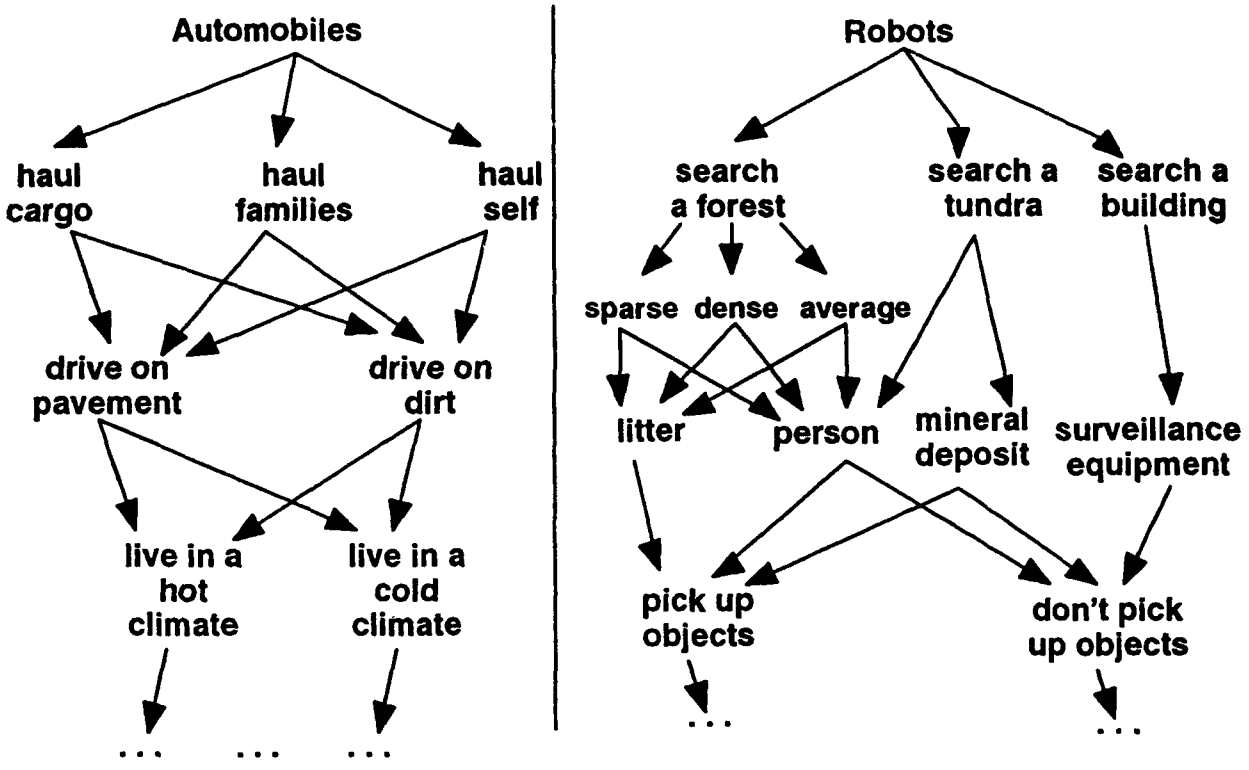
- What the problem has in common with other, similar problems
- How the problem differs from these other, similar problems

An application engineer precisely states the problem in terms of the domain by:

- Deciding how the problem differs from other problems in the domain. These decisions will require engineering judgment in addition to cold, hard facts.
- Validating the problem statement (i.e., the requirements) to make sure they precisely express the behavior the customer intended.

The following decision trees show part of the decisions needed to identify how problems differ in the automobile and robot domains.

Example Decision Trees for Precisely Stating the Problem



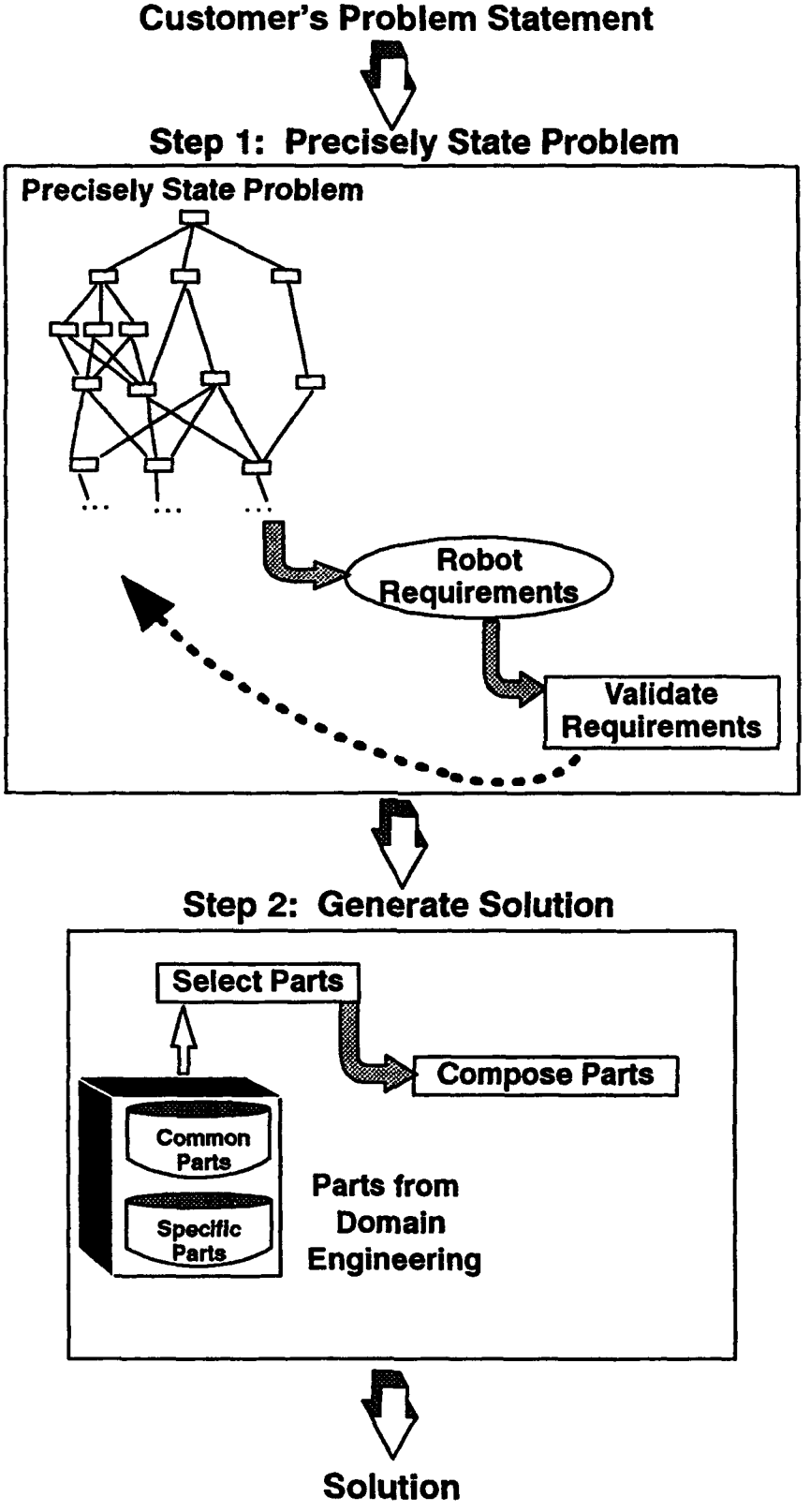
Step 2: Generate Solution

The application engineer then generates a solution based on the precise problem statement from Step 1. To do this, the application engineer uses the application engineering environment set up by the domain engineer. This environment contains:

- Software components needed to generate a solution to a problem in the domain. These components include:
  - Components that are common to all solutions
  - Components that solve only specific problems
- Help for how to put all of these components together to form a solution.

The following figure represents what happens in the two steps of application engineering.

APPLICATION ENGINEERING:



This page intentionally left blank.

UNIT 3: APPLICATION ENGINEERING LABORATORY

PART 1: BACKGROUND

In this laboratory, you will practice application engineering. Imagine yourself to be an application engineer who works for URW. Three customers approach you. Each has different needs:

1. Customer 1, a farmer, owns a large cornfield and has trouble finding time to harvest it. She wants to know if you can provide a robot that will harvest her corn without human supervision.

2. Customer 2 is from the Alaska National Guard, which is constantly rescuing people who wander too far afield in the tundra. Mounting a rescue party is time-consuming; people have died while the members of the party were gathering. The Guard thinks having robots ready could eliminate these life-threatening delays.

3. Customer 3, from the National Park Service, is concerned about growing amounts of litter in national parks, and wants to know if you can provide a robot that can pick up the litter.

These three statements correspond to customers' vague understandings of their problems and of potential solutions. Your task in this laboratory is to help these customers understand their problems fully and to provide them with robots that solve their problems. To assist you in this, we have provided you with a tool that automates some of the application engineering. Part 2 describes its use.

In brief, you will be asked to generate the software for a robot. You will do so by following the application engineering process for precisely stating a problem, which you saw in class. Some of the decisions you must make can be answered from the three statements above. Others may require clarification from your customer. Your instructor will act as the customer, answering questions you might have on the requirements for the robot. Keep in mind, though, that a customer does not necessarily know everything. As an application engineer, you are expected to use your own expert judgment when your customer does not know what choice is right.

PART 2: EXERCISES

1. CORNFIELD ROBOT

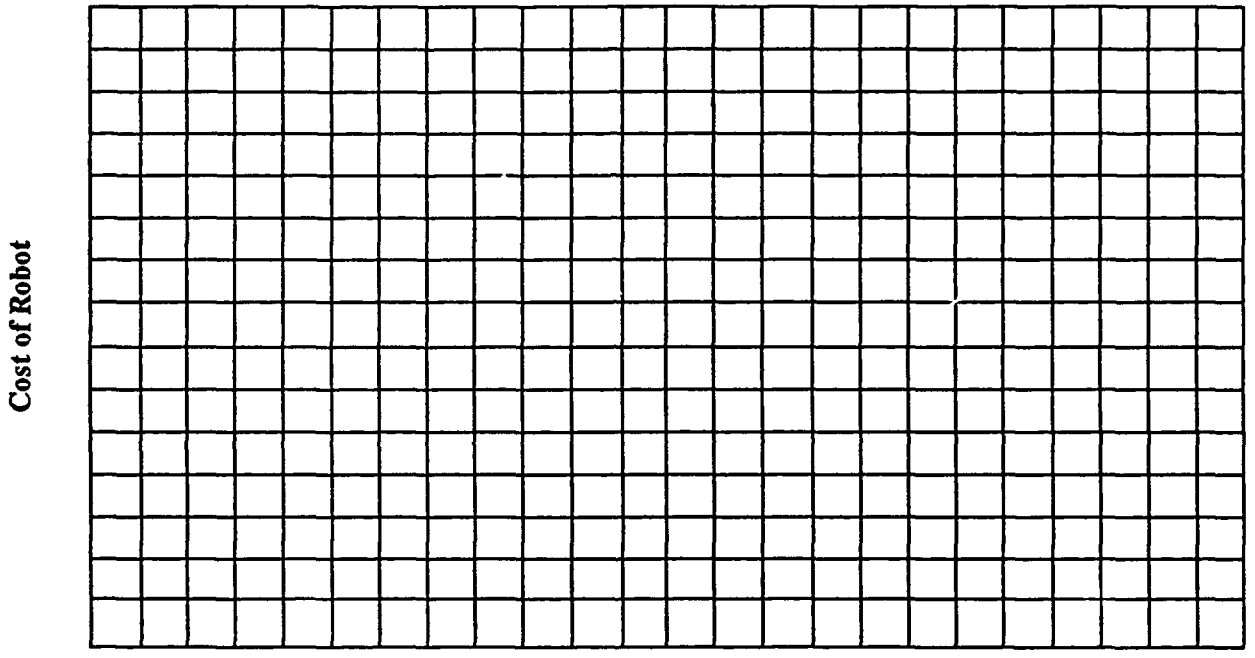
URW manufactures robots that can harvest corn. You must act as an application engineer and help solve your customer's problem by resolving the decisions in the domain. By doing so, you will create a model of a robot that harvests corn. You can use this model to generate the software that controls the robot. However, you cannot just generate any corn-harvesting robot. In the first place, your customer has a specific requirement: she wants the robot to end its mission at its point of origin. In the second place, she cannot spend more than $13,500.00. The robot you model must not exceed this price. Better still, it must be the least expensive robot that can do the job.

You will be informed of the robot's price as part of validation. However, you should know that two factors determine a corn-harvesting robot's price. The first factor is the maximum number of ears of corn it can carry. URW offers its customers robots that carry between 50 and 500 ears, in multiples

of 10. (The decision to carry 53 ears therefore results in a robot that costs the same as one that carries 60 ears, although the former robot will still pick up, at most, 53 ears.)

The second factor is the number of batteries with which the robot is equipped. All robots have at least one battery. Each extra battery costs money, but increases the distance the robot can travel. To estimate the **minimum** number of batteries needed, press the `F1` key when you are asked to make the decision on the number of batteries.

Question A. Find the most appropriate robot for your customer. Do so by repeating the process of precisely stating the problem, varying the decisions until you believe your model of robot is right. Use the following graph to correlate the cost of each robot to its carrying capacity. What trend do you observe?



Question B. Generate and execute the software for three robots: the one you described in Part A, the one with the minimum carrying capacity, and the one with the maximum carrying capacity. Record the time needed for each one to execute. Which takes the least time? Would you have created a different robot for your customer if time for harvesting had been her highest priority?

2. Rescuing Robot

Generate a robot that meets the needs of your second customer. What decisions related to choosing the mission are clearly invalid? Why?

Run your robot several times. Notice that there is more than one tundra for your robot to search. Compare their characteristics. Which one requires more energy? Would you recommend that your customer equip his robot with enough batteries to handle either case, or do you think your customer would be satisfied with a less expensive robot that could only handle the low-energy case?

3. Litter-Gathering Robot

Generate a robot that meets the needs of your third customer. For this customer, you will find that you need to try more than one robot to determine which one is best. The reason is that URW has two searching strategies for robots that operate in a forest. One is to "sweep" back and forth, horizontally; the other is to zigzag. Which is more effective depends on the forest in which the robot is operating.

Precisely state the problem for this robot. Use the results to fill in the following table:

Density of Trees in Forest	Cost of Robot	
	Search by Sweeping	Search by Zigzagging
sparse		
average		
dense		

What do you observe about robot cost versus forest density?

PART 3: USING THE APPLICATION ENGINEERING ENVIRONMENT

This part of the laboratory describes how to use the application engineering environment for specifying and generating robot software. Your instructor will tell you how to invoke the environment. Once you have done so, you will see the following menu (the main menu):

```
APPLICATION ENGINEERING ENVIRONMENT
              FOR
         ROBOT DOMAIN

   1) Precisely State Problem
   2) Generate Solution
   3) Execute Solution
   4) View Generated Software


   Select an item:
```

You will follow the application engineering process by selecting each of the first three menu items, in the order listed. Item 1 assists you in precisely stating the customer's problem—that is, decision making and validating the problem statement. Item 2 generates a solution based on your statement of the problem. Item 3 allows you to simulate execution of a robot, using a modified version of the Karel executor program (please note that the programs you generate will not work with the Karel compiler or executor you have used previously). Item 4 lets you see the software you generate using Item 2.

To select a menu item, type the number of the item, followed by the ENTER key. If you need help, press the F1 key. You can use the backspace key to remove the last character you typed. When you are finished, press the F2 key to exit. (These statements apply throughout the application engineering environment.)

Precisely Stating the Problem

Selecting Item 1 from the main menu gives you the following menu:

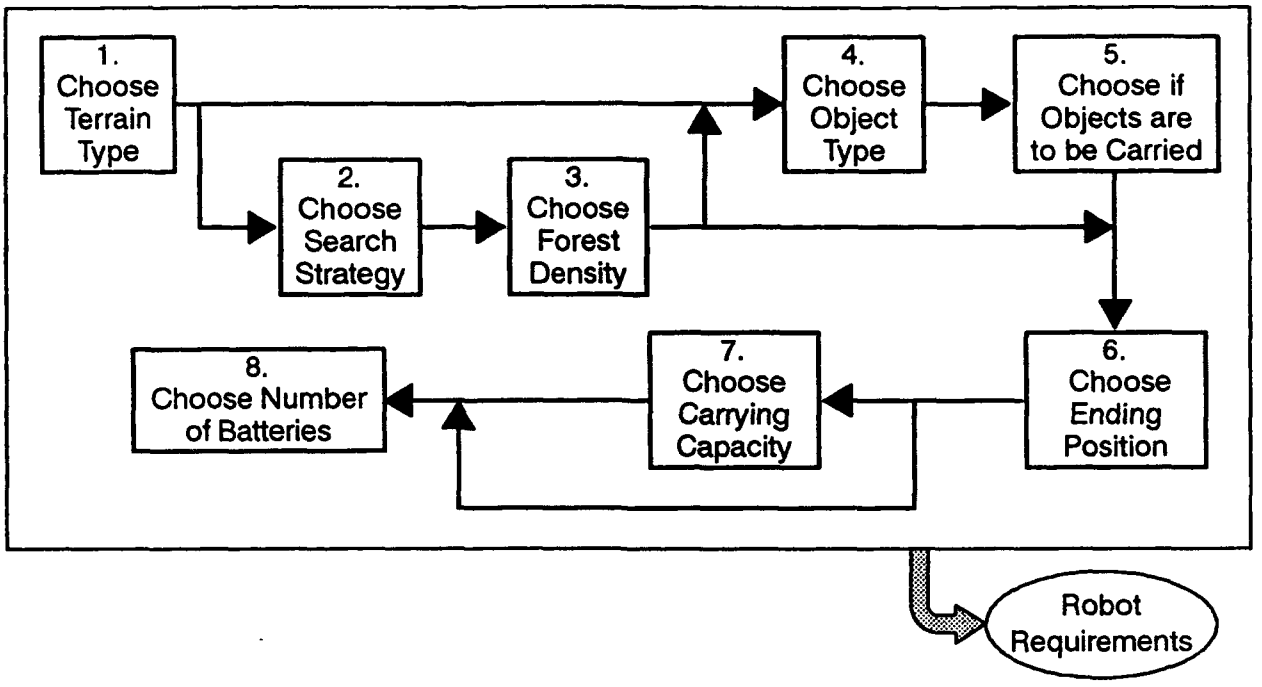
```
         PRECISELY STATE PROBLEM
                   FOR
              ROBOT DOMAIN

   1) Make Decisions
   2) Validate Statement of Problem
   Choose a step:
```

Use this menu to make the decisions needed to identify a robot that meets a customer's needs (Item 1) and to validate the decisions you have made (Item 2). Once you have performed these two steps, press F2 to return to the main menu.

Making Decisions

Selecting Item 1 from the menu for precisely stating problems lets you step through the decision-making portion of the application engineering process shown in class:



You will be presented with a screen divided into three windows. The upper left window shows the decision you are making and the values you can choose for that decision. The upper right window shows all decisions, including the values of those you have made so far. Each time you make a decision, you will see the implications of that decision in the lower right window.

You will make the decisions in the order shown in the picture. In most cases, you will be given a menu and asked to choose an item. Enter the number of the item. For Decision 5, you will be asked a yes-or-no question; give the full word as an answer, not just Y or N. For Decisions 7 and 8, you will be asked to enter an integer value. Remember to follow your answer by pressing the ENTER key. When you have made all the decisions and think they meet your customer's requirements, press the F2 key. You will return to the main menu.

You can abort the decision-making process by pressing the ESC key. If you abort the decision-making process, you must make all the decisions again.

You can use the up and down arrow keys to move among the decisions. You can use this feature to examine subsequent decisions you must make or to change a decision you have made. Keep in mind that you must always make decisions in the order shown on the screen in the upper-left window. If you change a decision, all decisions following it need to be made, even if you already made them.

Each time you make a decision, you will be shown the implications of that decision. These implications are presented in terms of how they affect the robot's hardware and software. You can feel free to experiment with different combinations of decisions. You should be able to see how different customer needs result in different robots.

When you make Decision 8, you will probably need help estimating how many batteries your robot will require. Press the F1 key, and you will be shown some values. Bear in mind that these are estimates. Depending on the nuances of the terrain in which your robot operates—specifically, the

distribution of objects and obstacles—your robot may actually need more or less energy for a particular mission. Keep in mind the consequences of failure as you choose the number of batteries. A robot that runs out of energy before picking up all litter is a nuisance. A robot that runs out of energy before reaching a stranded party of hikers can have tragic consequences.

Validation

Once you have made all decisions, you are ready to validate your problem statement. In fact, much of the validation is already done. During decision making, you could not state a carrying capacity for a robot that does not pick up objects—the application engineering environment will not allow it.

However, you might still have made mistakes. For example, you could have misunderstood your customer's requirements and how they relate to the decisions. During validation, you are asked to review the decisions you have made. This is the time when you should make sure they are proper. To validate your decisions, select Item 2 from the Precisely State Problem menu.

During validation, you are also told how much the robot will cost. Unless your customer has a very deep wallet, you should check to make sure that the robot's price is within the customer's range.

If you decide that your decisions are improper, you can easily revise them. Exit validation, and choose Item 2 (Make Decisions) again. This time, you will see the decisions you made previously rather than a set of decisions waiting to be made. You can use the down-arrow key to move directly to the decision(s) you want to change.

When you think you have a valid set of decisions, press `F2` when you see the Precisely State Problem menu displayed. This will return you to the main menu.

Generating a Solution

Once you are satisfied with your statement of the problem, you can generate a solution to it. Just select Item 2 from the main menu. The program will choose all the correct parts for your solution and assemble them into a working program, which it will then compile for you. If you would like to examine the software you generated, select Item 4 from the main menu. You are now ready to simulate the execution of your robot.

Simulating Execution

Choose menu Item 3 from the main menu. This invokes the modified Karel executor mentioned earlier. Unlike the simulator you may have used, the program to execute and the map are chosen for you automatically. (After all, you wouldn't want to run a robot meant for a cornfield through a forest!)

Prior to execution, you will be presented with a set of questions that control how much information you see during execution. Be aware that this information, although interesting, can add up to 15 minutes to execution time. Moreover, you do not need it to complete the laboratory. You should opt not to display it if you are pressed for time.

UNIT 3: APPLICATION ENGINEERING LABORATORY

TEACHER NOTES FOR LABORATORY

Comment: This laboratory lets students use an application engineering environment. The environment implements the application engineering process for the robot domain covered in the Unit 3 lecture.

The environment plays down the role of programming. Students create programs, but not as they have previously. Instead, the environment automatically creates the software based on a problem statement the student provides. Theoretically, students can perform this laboratory without ever seeing any software. The environment contains a tool that lets them do so; this emphasizes that software is necessary to the robot but that it can be developed in more than one way.

What substitutes for programming is:

- Eliciting and understanding customer requirements and elaborating them in terms of the domain problem space. The result is a precise problem statement.
- Quantitative and qualitative analysis of requirements to determine satisfaction of customer needs.
- Simulation as a means to validate customer requirements.

The second item is most significant and probably less intuitive to students than the others. Students are asked to study problems and certain properties of solutions purely in terms of domain problem space concepts. They are not allowed to think in terms of primitive Karel instructions or even algorithms. They must act as application engineers, not programmers. By having them do so, you can demonstrate to them that programming is only a means to an end, not an end in itself.

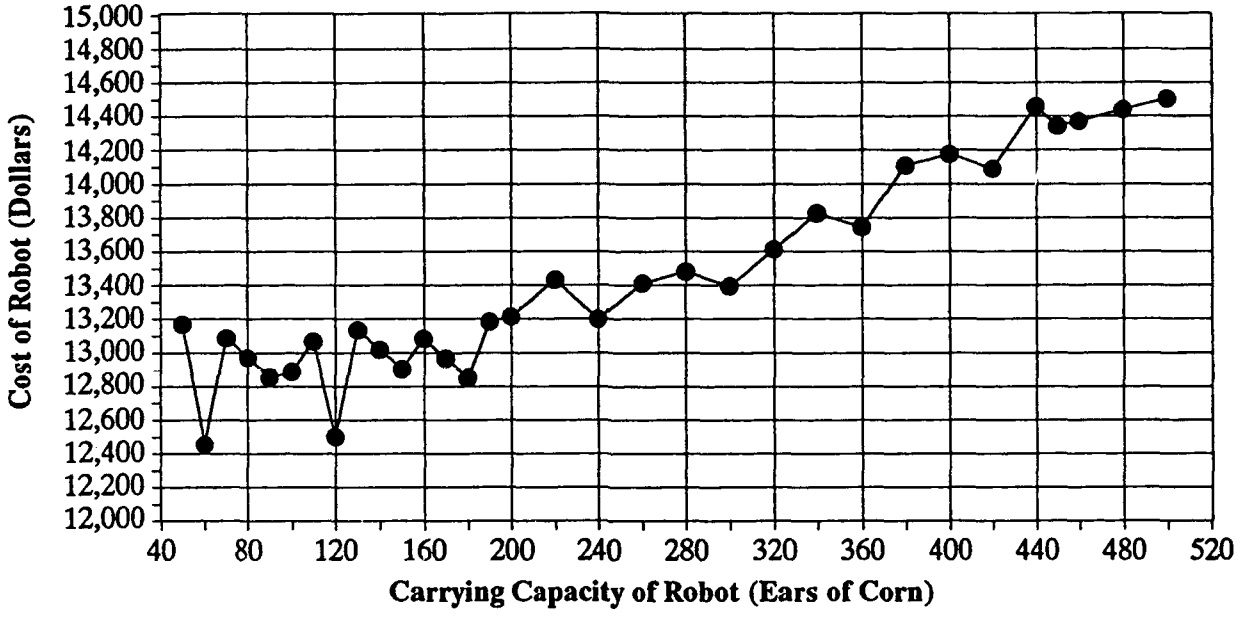
1. CORNFIELD ROBOT

Answer to Question A: This question asks the student to analyze a problem without first trying to generate a solution to that problem. The application engineering environment presents all the information the student needs. The student must first precisely state the problem, selecting "field" as the terrain; this fixes the decisions on search strategy and object type and obviates the decision on forest density. If any students wonder why, you can explain it to them as decisions already made by the domain engineers:

- In fields, URW only knows how to build robots that harvest corn. It doesn't possess the technology to build robots that mechanically harvest, for example, tomatoes.
- The domain engineers' studies have concluded that sweeping is a more efficient strategy than zigzagging when harvesting corn. (Real harvesting machines work this way.)

The student must choose the most appropriate robot. The assignment defines this as the robot that costs least, but can still perform its mission. Since carrying capacity and number of batteries are the

two factors that determine a robot's cost, the student must experiment with variations of these quantities to complete the assignment. The students will simply have to try several values of carrying capacity. They can determine the number of batteries through the help facility (available by pressing the `F1` key). During validation, they can obtain the cost of the robot they have modeled. Using this information, they should create a graph similar to the following:



Inform the students that they will need some strategy for choosing carrying capacities, unless they are so motivated as to try all 451 possible values. Note the trend of cost increasing as a function of carrying capacity. This should motivate them to try a binary search strategy. Binary search by itself is not adequate, because the robot's cost does not increase monotonically as a function of carrying capacity, but it is a good start.

For this mission, the robot costs least when its carrying capacity is 60. Here is the reason why. In the cornfield, each location contains one ear of corn. Therefore, each row has 30 ears. Any multiple of 60 minimizes the number of spaces a robot must move to unload its cargo and return to continue harvesting, since it always fills its bag when it is against the western border. A value that is not a multiple of 60 would require the robot to move west as well as south as it returns. Each move consumes battery power, necessitating extra energy; since multiples of 60 minimize moves, they are preferred. Note that the robot will make fewer moves if its carrying capacity is 120 instead of 60, and indeed will make the fewest moves if its carrying capacity is 449 (the number of ears of corn in the field). However, extra carrying capacity costs money, and carrying 400-plus ears increases the robot's weight enough to cause it to consume energy rapidly. This in turn requires extra batteries, driving up the robot's price. For these reasons, 60 is the optimal carrying capacity.

This fact—that robots in cornfields behave best when their carrying capacity is a multiple of 60—is an excellent example of the type of knowledge possessed by experts in a domain. That is, it is something an application engineer would know and would automatically apply when approached by a customer. This knowledge would be gained by experience, through trial and error. Deriving it mathematically is difficult; in many domains, it is impossible. Eventually, application engineers feed this type of

trial-and-error experience back into domain engineering, where experts incorporate it as a heuristic in the application engineering environment.

You can discuss this with students. Ask them for commonplace but significant knowledge in other domains. A few examples: does your automobile owner's manual tell you how to park your car? Few do; of those that do, do any tell you to put money in the parking meter? Does your owner's manual say to turn off your ignition after you park your car?

Answer to Question B: The following are some sample results:

Carrying Capacity (Ears of Corn)	Execution Time (Seconds)
50	1:00.25
60	46.14
500	33.84

These numbers were obtained running the Karel simulator on a 486-based computer. The numbers you obtain will depend upon the computer you use. However, you should still obtain the same ordering: a carrying capacity of 50 results in the slowest execution time, and a capacity of 500 results in the fastest. Therefore, if your customer wants a robot that can harvest corn as quickly as possible, and if money is no object to her, you should recommend that she choose the robot with the greatest carrying capacity. The most alert student will also observe that, as a field contains at most 449 ears of corn, the customer could save a little money without sacrificing execution speed by buying a robot whose carrying capacity is 449 or 450 (both these robots cost the same).

To obtain consistency in the results, the students' answers to the questions asked by the simulator must be identical for all three trials. Be aware that the executor can run very, very slowly. It is usually best to answer N to the three yes/no questions (see Simulating Execution on page 10), and to set the speed to 0. You can use this as an opportunity to reenforce experimental science concepts to your students.

You are not actually running a robot; you are running a simulation. If URW were a real company, the application engineer would run a simulation such as this to learn facts about the robot's performance that cannot be determined in other ways (i.e., as part of validation). This point is well-illustrated in laboratory Questions 2 and 3, with their somewhat randomly-placed objects and obstacles. Addressing the issues raised by Questions 2 and 3 by deriving formulas is very hard. Simulation provides a simpler alternative.

2. Rescuing Robot

Answer: There is no point in deciding that a robot should pick up hikers and continue until it runs out of energy. The purpose of a "rescue" mission would be either to transport the hikers to a safe, known place (either the origin or the point where the entire terrain has been covered—both can be predicted) or to stay with the hikers until help arrives. If the robot continued until it ran out of energy, the National Guard would have difficulty locating it, so the hikers would be no better off than if they had just stayed where they were.

This laboratory comes with two maps of a tundra. One, named tundra1, is intended to illustrate the average case. A group of three hikers is stranded more or less in the middle. The other map, named

tundra2, illustrates the worst case. There are a total of five hikers (the maximum permissible carrying capacity). Four are right at the beginning of the robot's search. The remaining hiker is at the very end. Suppose you opt to have the robot pick up hikers and return. The robot will consume the maximum possible amount of energy. It must carry four hikers the greatest possible distance before it completes its search by finding the fifth. Since carrying an object consumes energy, the robot's energy use is maximized.

Choosing instead to have the robot stop when it locates a person creates a robot that is probably unsatisfactory. It will find one group of hikers but not the other. This is not likely to please the National Guard, nor is a robot with a carrying capacity so small that it returns before it finds everyone.

The purpose of this question, then, is to make sure the students study the problem carefully and truly understand the needs of their customer. They must pay particular attention to the following:

- Only certain combinations of ending location and carrying capability are useful for rescuing people.
- The robot must not run out of energy. In a cornfield, the consequences of doing so are annoying. In a tundra, human lives are at stake. Failure has dire consequences.
- Application engineers must make important choices based on their own judgement. The application engineering environment cannot calculate the right amount of energy. It can predict average use (note that the robot will actually fail if given the average number of batteries needed: the hikers are just a bit beyond the midpoint, which is assumed to be average), and it can predict worst-case use. The worst-case robot works but is very expensive. Most customers are not willing to pay the price on the off-chance that the worst case will occur. They want something that handles most cases. The application engineer has the moral responsibility to present this information to the customer and to try to come up with the best energy statement. In the laboratory, you might want to act as customer and establish an arbitrary price ceiling that precludes building the worst-case robot. As part of the assignment, ask the students to prepare a report of what they expect the robot can do.

One note: the simulator chooses one of the two maps used at random. In a class of 20 people, you can be 95% confident that at least one person will not see both maps even if everyone runs the simulation 4 times. Be prepared to ask students to keep running the simulator until they have used both maps. (The simulator shows the map's name in the lower left window.)

3. Litter-Gathering Robot

Answer: The following table was created using a carrying capacity of 250, with the robot picking up litter and returning to its point of origin when its bag is full. The average-case number of batteries was used. Such a robot does not have enough energy to complete is mission, but the general trend illustrated by the table does not change with the number of batteries.

Density of Trees in Forest	Cost of Robot	
	Search by Sweeping	Search by Zigzagging
sparse	$9,944.00	$11,806.00
average	$10,368.00	$11,673.00
dense	$10,875.00	$11,540.00

Notice the difference between the columns. The cost of a robot that sweeps is proportional to the forest density. The cost of a robot that zigzags is inversely proportional to forest density. If you examine the code, you will observe that navigating around a tree in a sweep requires two extra moves and eight extra turns. By contrast, zigzagging around a tree requires four fewer turns than if the tree were not present. In theory, then, a robot moving in an extremely dense forest (or a larger one) would do better to zigzag. In practice, a Karel map cannot contain enough trees to make this worthwhile.

This page intentionally left blank.

START OF FOURTH UNIT

DISCUSSION

This unit describes the concepts underlying the tools used in the laboratory exercise.

Domain engineers develop a "black box" (in fact, you used a black box in the laboratory exercise); the application engineer uses the black box when creating a solution for a customer. The application engineer doesn't need to know what is inside the black box to use it.

Unit 4 opens up some of the black box. It gives a brief introduction to the major products created during domain engineering:

- The application engineering process
- The description of what problems are in a domain
- The domain architecture
- The architectural components

The unit cannot open up all of the black box. It deliberately avoids issues of representation, because of time constraints. It does not attempt to cover how to do domain engineering; it discusses only the results and how they relate to application engineering.

This unit relates megaprogramming to the skills that the students have already acquired in class (that is, domain engineering requires programming skills).

OBJECTIVES FOR THE ENTIRE UNIT

The students should be able to:

- Understand some of the concepts underlying the tools used in the Unit 3 laboratory
- Relate megaprogramming to the programming skills they have acquired

Unit 4: Domain Engineering

DISCUSSION

This slide shows that application engineers produce robots through a process supported by a **black box** called **process support**. The laboratory from Unit 3 required students to use this black box.

Therefore, we can think of application engineering as requiring:

- A process (the steps for producing robots and the robot software)
- Process support:
  - Karel instructions (software procedures and functions)
  - Automated tools that support the process (like the programs in the laboratory that helped you state problems and generate solutions)

The laboratory showed how use of these made producing robots straightforward.

Process and process support are created during domain engineering. How did they come to be?

- How did domain engineers decide what the process to precisely state the problem should be?
- How did the domain engineers decide what process support to build?
- For that matter, how did domain engineers decide what the domain should be? Why did they include some robots and not others? For instance, why were only three terrains allowed?

Let's look at each of these questions, starting with the last one.

STUDENT INTERACTIONS

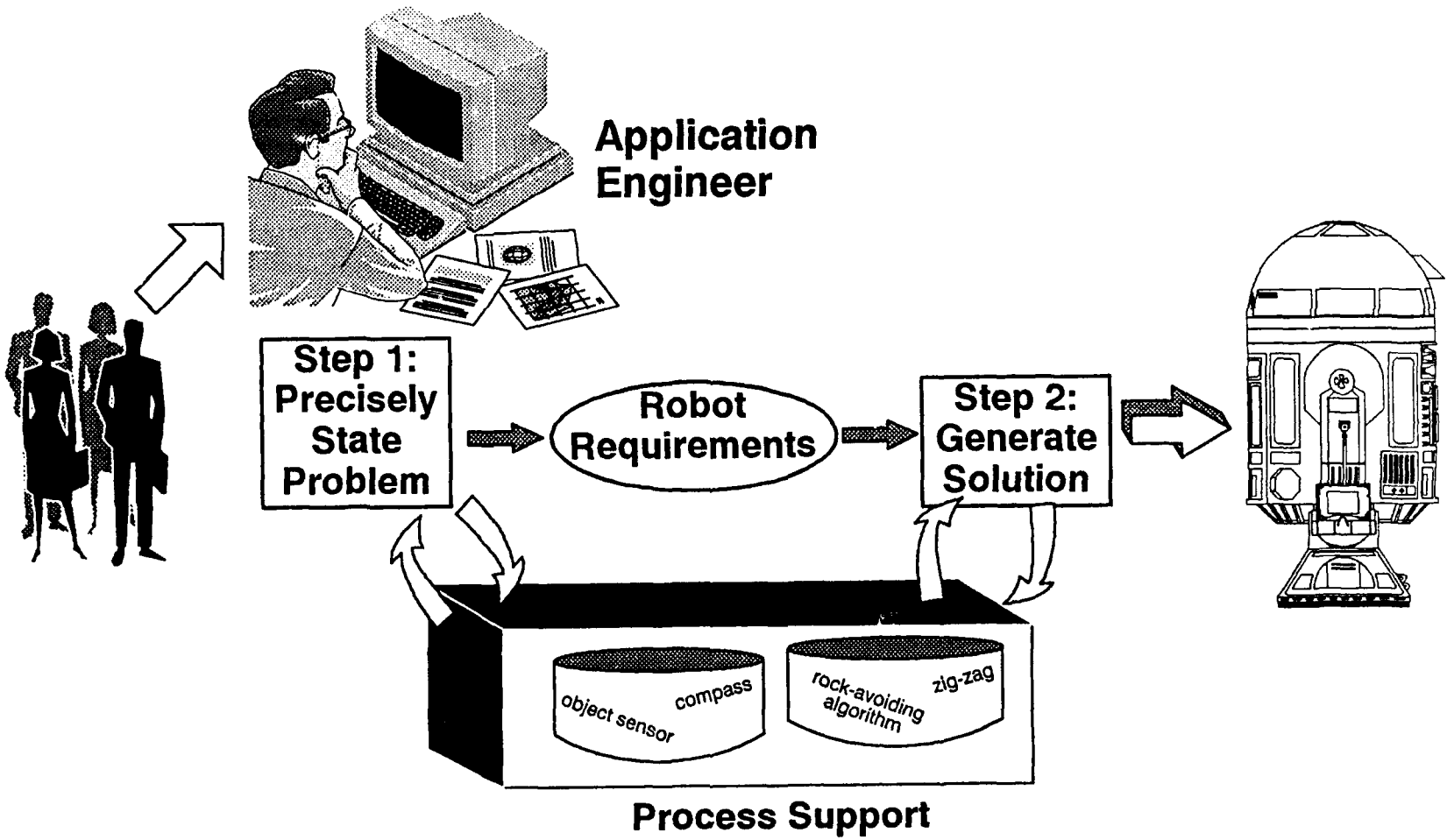
- Do you think process and process support exist in other domains? (A **very** good example is a fast-food restaurant.)

OBJECTIVE

The students should be able to:

- Explain the application engineering process

What Help Do Application Engineers Need?



DISCUSSION

This slide presents examples of factors (in terms of our robot domain) that domain engineers consider when they decide what problems and solutions to include in a domain.

It's domain engineers, not application engineers, who decide what's in a domain. Application engineers create individual systems. Domain engineers are responsible for deciding the range of systems that application engineers can create.

In our robot company, the domain engineers might think along the following lines:

- We know how to build robots that operate autonomously, and we think our customers would be happier if they didn't have to control a robot as it operates.
- We think that the demand for robots that search for objects is larger than the demand for robots that place objects (an example of the latter is a robot that sows seeds).
- We've built robots that operate in a tundra, a forest, and a field. We can continue to do so without investing lots of money.

These are the types of factors that constrain the problems and *solutions* that form the domain (notice there are both business and technical factors). Part of domain engineering is studying these factors and deciding what problems and solutions are important for a successful business.

Once the domain engineers know the problems that will be in the domain, they can study them and uncover the commonalities among all problems and the differences between instances of problems.

This is the basis for the decision-making process we saw in the Unit 3 laboratory. What sort of information helps domain engineers create this process?

STUDENT INTERACTIONS

- How do General Motors, Ford, and Chrysler decide what types of cars to build (that is, what kind of cars will be in their domain)? Are their domains completely the same or are there differences?

OBJECTIVE

The students should be able to:

- Explain the types of information domain engineers use to determine which problems (and which solutions to those problems) should be in a domain and which should be excluded

Factors Influencing What Is In a Robot Domain

- The knowledge about robots you possess
- The types of robots your company has built
- The types of robots you think your customers want
- The types of robots your boss wants you to build

DISCUSSION

Domain engineers create a decision-making process based on what's common and what's variable in the domain. They deliberately exclude from this process all the things that are common—if something is common to all problems and solutions, there's no decision to be made. For example, you weren't asked in the lab if your robot needed a locomotion mechanism because all robots need a locomotion mechanism.

This slide, then, shows the type of information that helps domain engineers to reason, systematically, about a process for describing robots. They will later use this information to create a process that the application engineer can follow.

This is the type of information that you would know if you were familiar with the domain. (In fact, most of it makes good sense even if you're seeing the domain for the first time.) Domain engineers are familiar with the domain. They would have no trouble listing facts such as these.

The list is not ordered. Domain engineers create such lists using stream-of-consciousness thinking. They must subsequently organize it, but at first they simply try to enumerate their knowledge of the domain.

Remember again: this information emphasizes what varies between robots. Making decisions is (by definition) based on variations, not on commonalities.

How do we use information of this sort to help us create a process?

STUDENT INTERACTIONS

- What is some of the information that could be used to create a decision-making process for choosing an automobile (e.g., color, four-wheel drive, number of doors, engine size, etc.)?

OBJECTIVES

The students should be able to:

- Explain the type of knowledge that goes into creating a decision-making process
- Explain that making decisions is based on variations, not commonalities

Information That Helps You Create a (Robot) Decision-Making Process

- Robots whose mission is to locate objects but not carry them do not need arms or bags.
- Some objects cannot be carried; they might be too big or immovable.
- A terrain contains certain types of objects; no terrain contains all types of objects.
- Some terrains are much larger than others; we would not expect a cornfield the size of a tundra!
- . . .

DISCUSSION

This slide covers issues in creating a process, emphasizing the decision-making portion of the process.

At a minimum, the process must ensure that application engineers make all relevant decisions, validate the decisions, and generate the software. Domain engineers use the information about robots (see previous slide) to formulate the decision-making process. In any domain, some decisions influence others. Here are two examples in the robot domain:

- Whether objects should be carried (Decision 5) influences carrying capacity (Decision 7).
- The terrain (Decision 1) influences the object type (Decision 4) and number of batteries (Decision 8).

The decision-making process is ordered to reflect these influences. (This is not the only valid ordering.)

You are used to validating your programs after you generate them. In our robot domain, you do it before! We are able to validate before generation, because we know that the decisions are an accurate model of the software. The earlier we validate, the less effort we waste if we make mistakes.

In the laboratory, most of the process was automated. A program asked you to make decisions in a particular order, helped you validate your requirements, and generated the software for you. Domain engineers should try to automate as much as they can. The application engineers provide feedback to the domain engineers on things that did and did not work with the process support. The domain engineers will improve the process support based on these comments. With many engineers (both domain and application) working to improve the process support, errors are more likely to be found quickly.

Through domain engineering, the domain engineer has made all these things pretty much "mechanical" and unambiguous. The domain engineer tries to provide **process support** that makes the computer do whatever is mechanical. Anything that requires creativity, the application engineer must solve independently.

Now, how do we go from the decision-making process to generating a solution?

STUDENT INTERACTIONS

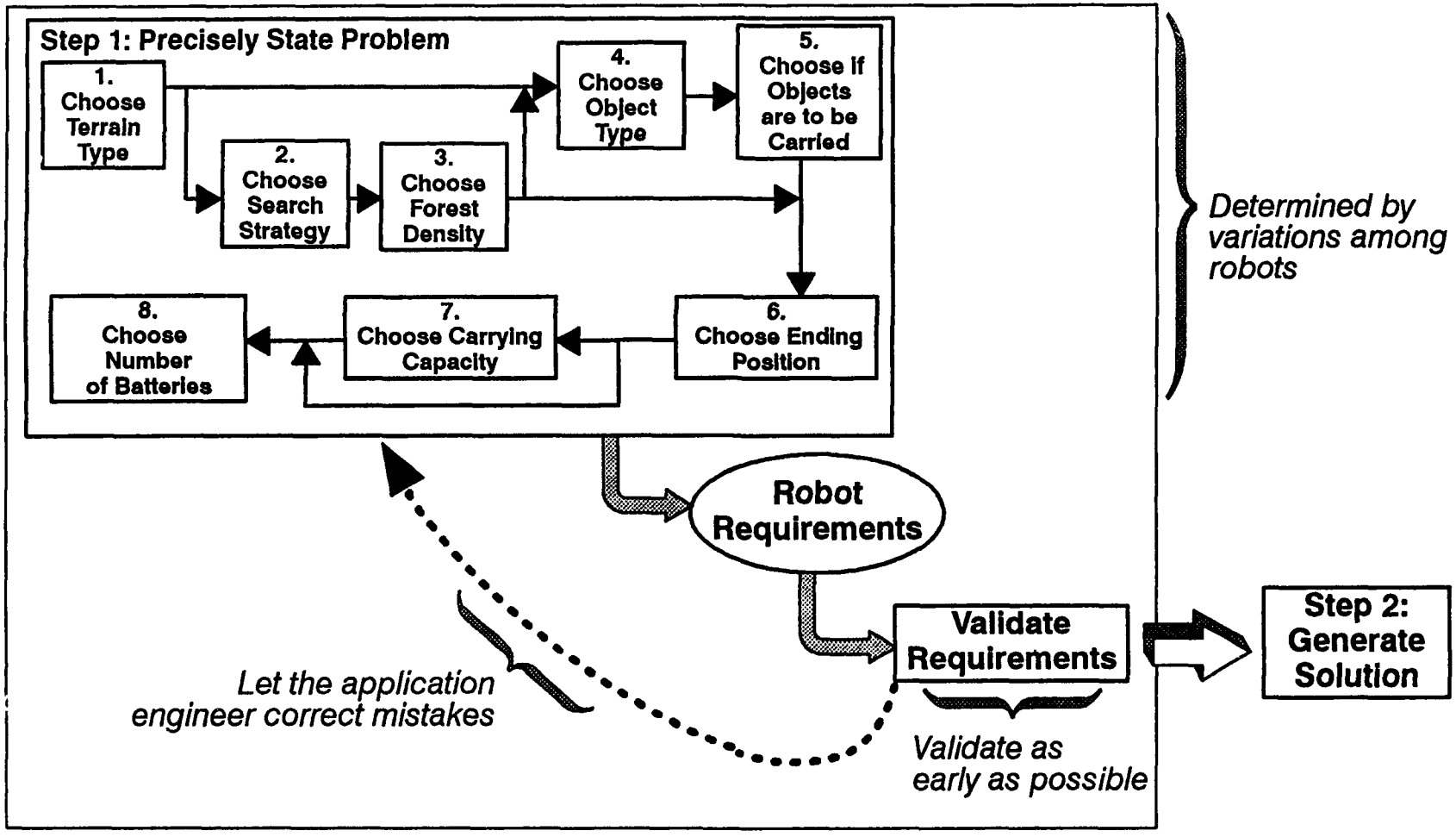
- Is the decision-making process mechanical for choosing an automobile? Could it be?

OBJECTIVES

The students should be able to:

- Explain the type of knowledge needed to create a process
- Understand the importance of automation in process support

Defining a Process for Precisely Stating a Problem



DISCUSSION

This slide begins a discussion (ending with 4-11) of how domain engineers define Step 2 of the Application Engineering process (generating a solution).

Domain engineers do three things to allow application engineers to generate a solution. First, they must develop an understanding of the structure of all programs for robots in the domain: what structural characteristics all programs have in common and how the characteristics vary. We call such a structure a **software architecture**. This slide presents, as an architecture, a calling hierarchy, showing the basic structure of robot software and suggesting some variation (e.g., forests and tundras have obstacles; fields don't).

Second, the domain engineers must develop **reusable code components** that can be used in this architecture. The architecture defines the Karel instructions needed, and their interrelationship, for a particular program. The domain engineer creates those instructions so the application engineers have them available when they want to create a robot.

Third, the domain engineers must define **generation procedures**. These describe how, for a particular problem, the architecture and the code components must be adapted to provide a solution to that problem. (Components as well as architecture must be adapted; for instance, Handle Object will need to be adapted to the type of object for which the robot is searching.) The generation procedures also describe how to fit the components into the architecture to form a complete program.

STUDENT INTERACTIONS

- Where do the reusable code components come from? Could they be obtained from existing robot software? Would this be of any benefit?

OBJECTIVES

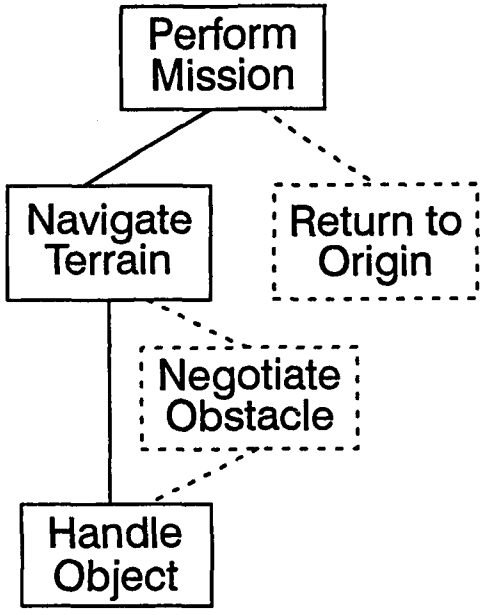
The students should be able to:

- State what the domain engineer must do to define how application engineers generate a solution to a problem in the domain
- Explain that every program has an architecture
- Explain that an architecture shows a program structure

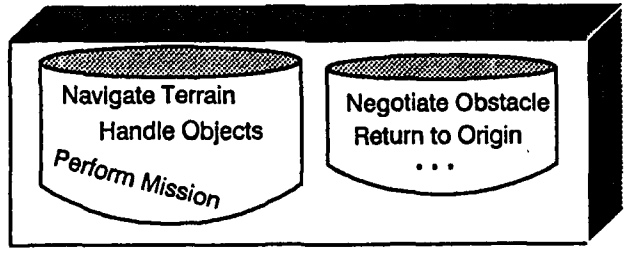
Defining How the Application Engineer Generates a Solution

The domain engineer must develop:

Robot Software Architecture



Reusable Code Components



Generation Procedures

- Always need to perform mission
- Always need to navigate terrain
- If terrain is forest, need tree negotiation
- If terrain is tundra, need rock negotiation

DISCUSSION

This slide begins a series that will show students how the domain engineer defines a software architecture for a domain. This slide begins by examining the architecture for a single program. Slides 4-8 and 4-9 are variations and generalizations of this one.

This slide is a detailed look at the software architecture of a particular robot, one which you were asked to generate in the Unit 3 laboratory (this should help you follow its logic). It shows the Karel instructions that make up the program (boxes), which instructions invoke which (lines), and the circumstances under which one instruction invokes another (commentary in italics). This slide shows only the major instructions (a slide with all instructions would be too crowded to be readable).

This architecture shows the general logic of the program. The main instruction is the one that drives the mission. This instruction does two things: it first invokes a zigzag toward the northeast corner of the forest and then invokes a return instruction. While zigzagging, it instructs the robot to pick up any litter. If it finds enough litter to fill the bag, it returns from that spot. This set of instructions is probably pretty similar to what you would write if your teacher asked you to create this one robot.

Now, how can we make use of this notion of architecture in building our robot domain? That is, how does an architecture help us generate a solution to any problem in the domain, not just to a single problem?

STUDENT INTERACTIONS

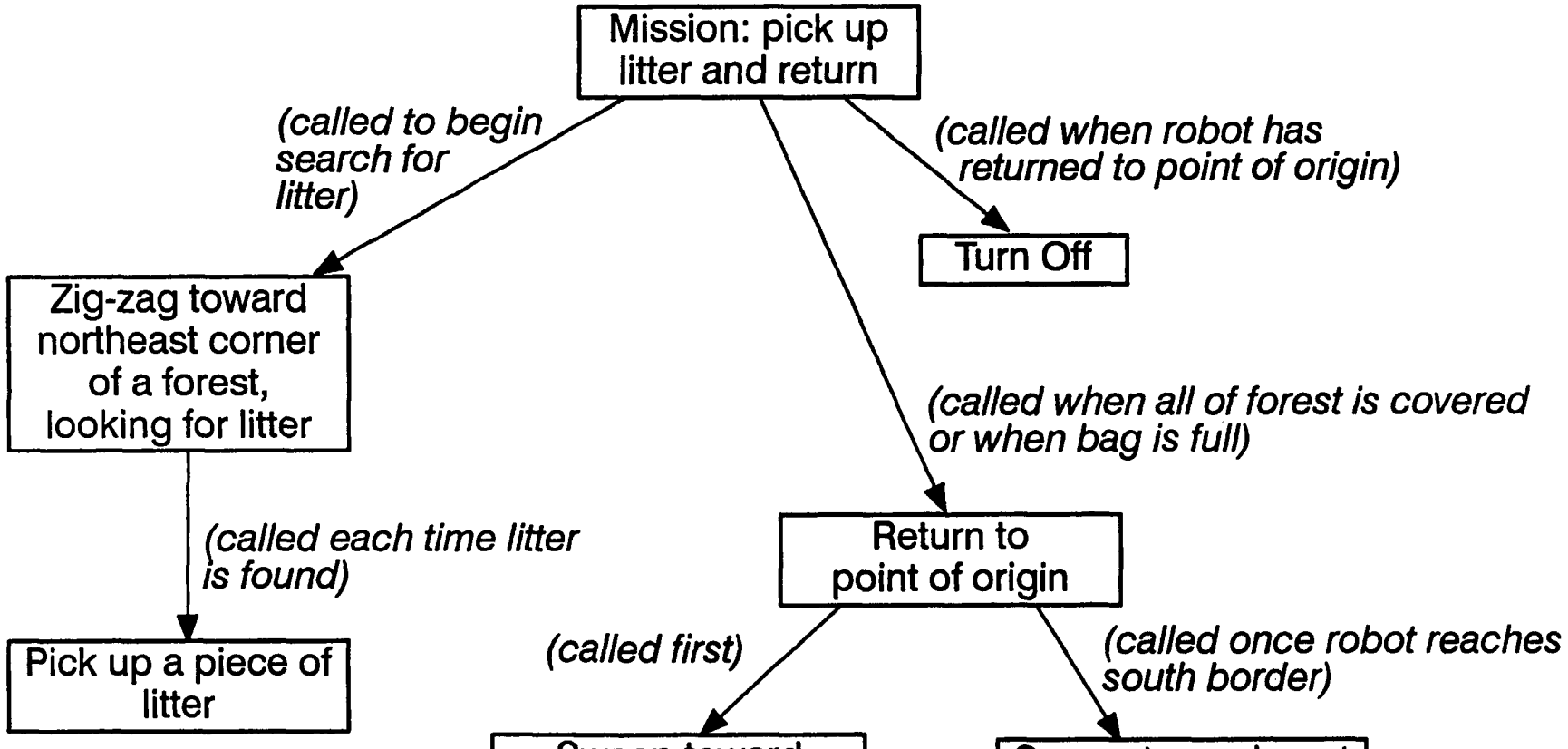
- Did every program you write this year have a software architecture? In what part of the software development process should you develop the architecture (requirements, design, code, or test)?
- Does a box correspond to a single Karel instruction? In Pascal, are data types part of the boxes?

OBJECTIVE

The students should be able to:

- Create a skeletal program based on an architecture diagram

Robot Software Architecture for Robot in Forest That Picks up Litter



DISCUSSION

This slide shows an architecture for a different robot than in Slide 4-7. Comparing and contrasting the two slides shows similarities and differences.

This robot searches a forest, but for hikers, not litter. It does not pick up hikers. Instead, it transmits their position when it locates them. Like the last robot, it returns to its point of origin when it reaches the northeast corner of the forest. Unlike the last robot, it does not turn off after returning. It sets out again, continuing this cycle until it runs out of energy.

This robot has an architecture similar to the previous one. For each box in one (except Turn Off), there is a corresponding box in the other. But note that on this slide:

- All boxes concerned with objects refer to people, not litter.
- The mission is different. This robot only searches for people. It does not pick them up. It searches indefinitely, not for a fixed period.
- A robot might transmit its position instead of picking up an object. This is a different reaction on locating an object.

STUDENT INTERACTIONS

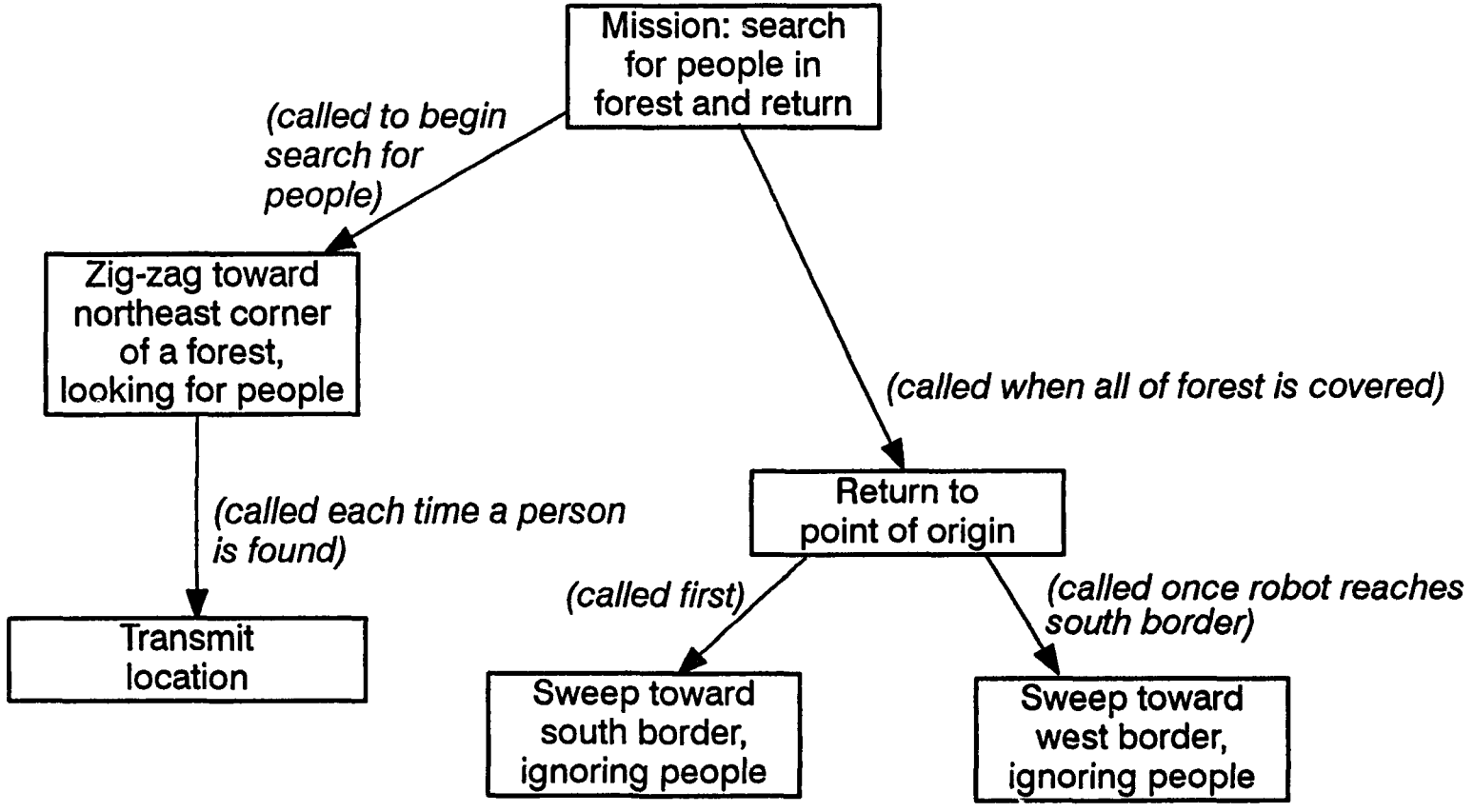
- Is there any difference in the strategy a robot would use to search a forest if it's looking for people instead of litter?
- Does this robot's software for sweeping differ from the sweeping software in the robot on Slide 4-7?

OBJECTIVE

The students should be able to:

- Explain the differences between this architecture and the architecture on the previous slide

Robot Software Architecture for Robot Searching Forest for People



DISCUSSION

This slide shows how two architectures may be combined into one architecture that can be adapted into either of the two original architectures. It combines the architectures of the previous two slides.

Domain engineers need to understand the similarities and differences among architectures. This knowledge lets them understand which parts of the software can be reused in different robots.

This picture shows some of the similarities and differences. Abstract phrases describe components:

- Every robot performs a mission, but we've seen that this mission differs from one robot to another. Let's generalize the top box to "Perform Mission."
- Different robots search for different objects. Let's remove the references to the type of object for which the robot is searching.

Also, comparing Slides 4-7 and 4-8 shows that some components aren't always needed:

- Robots return under different circumstances. In fact, some don't return at all.
- Robots don't always turn off voluntarily. Some operate until they run out of energy.

In this slide, dashed boxes mean the component is used in only some architectures. Arrows with a solid line mean the instruction at the tail calls the instruction at the head in all programs in the domain. Arrows with a dashed line mean the call exists in only some programs. (If it doesn't exist, the instruction at the arrowhead isn't needed.)

What is a general architecture for any robot in the domain?

STUDENT INTERACTIONS

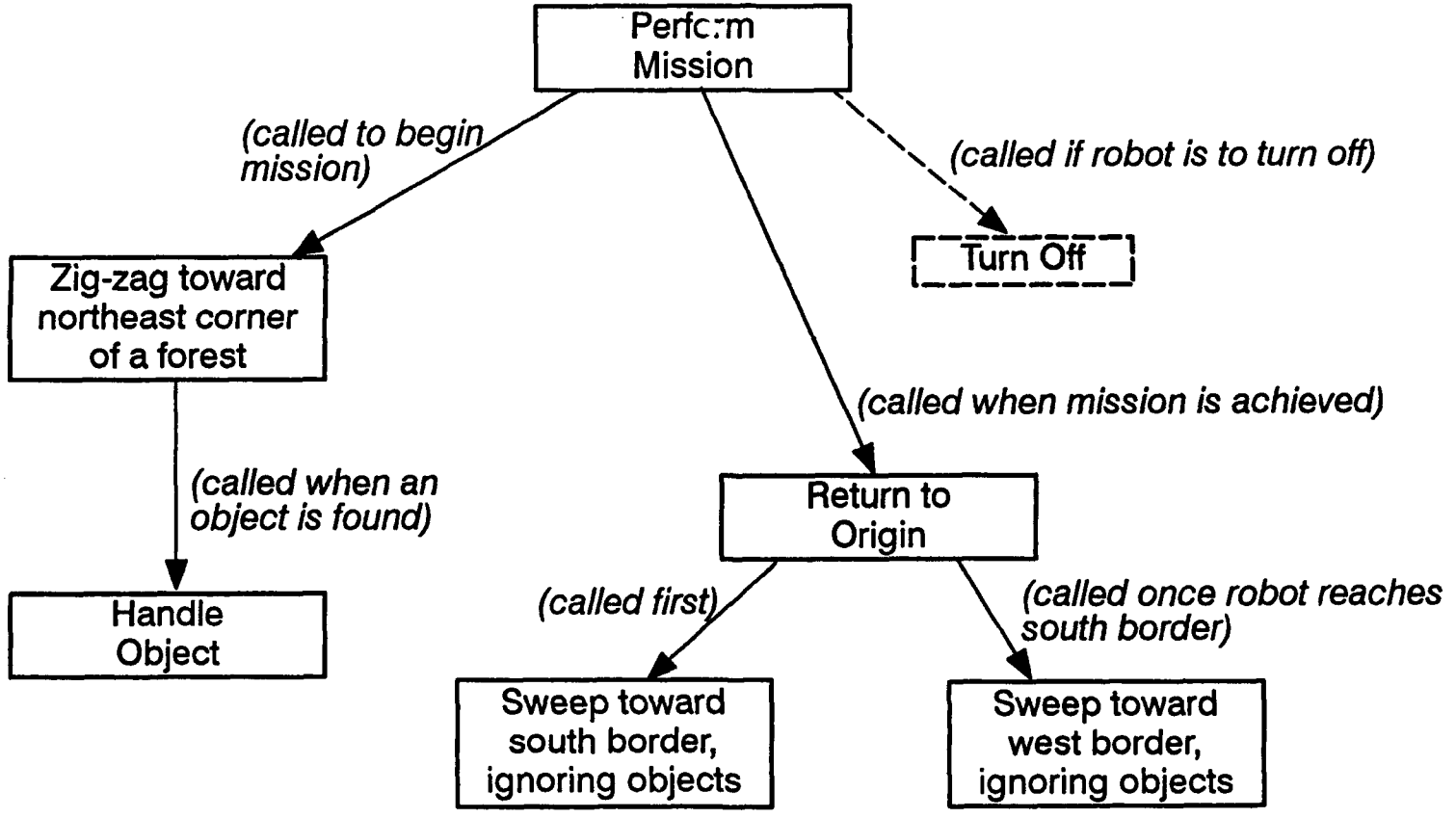
- Can you see how to get from this generalized architecture back to the specific architecture? Be as precise as possible.

OBJECTIVE

The students should be able to:

- Understand the transition from the specific robot architectures on the previous slides to the more general architecture depicted on this slide

Combining the Two Software Architectures



DISCUSSION

This slide completes the generalization of the architecture.

The architecture shown here is one that domain engineers agree can be used for any robot in the domain.

- All robot software is driven by an instruction to perform the mission. One instance of that instruction is "Pick up litter and return." Another instance might be to "Stop when an ear of corn is found."
- All robots need an instruction that tells them how to navigate through the terrain. In a tundra or field, the robot will sweep. In a forest, the robot will sweep or zigzag.
- Robots in fields don't have obstacles; therefore, they don't need any instructions to negotiate their way around them. Robots in other terrains must face rocks, trees, etc.
- Some robots return; others do not. Therefore, not all robots need an instruction to return.

Each box corresponds to a reusable software component. An application engineer builds robot software by selecting:

- Each component required by all robots in the domain (here, "Perform Mission" and "Navigate Terrain"), adapted to the nuances of the robot (e.g., "Pick Up Litter and Return" for "Perform Mission").
- The proper set of components that are needed to support the particular robot the application engineers want, adapted to the mission (e.g., "Negotiate Rock" for "Negotiate Obstacle").

The domain engineers have to figure out which reusable components are needed for which robots. That's not hard—it's just a matter of figuring out the right combinations. These are the generation procedures mentioned on Slide 4-6.

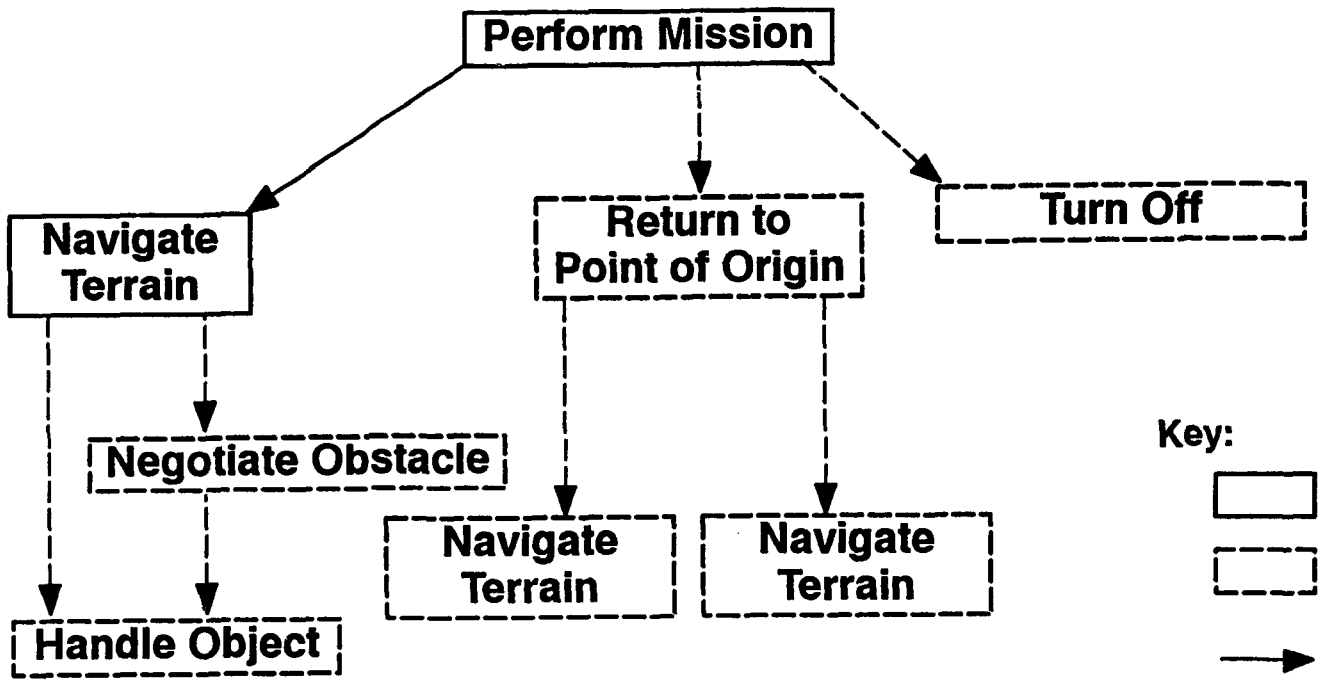
The domain engineers also have to create the reusable components. This is where all the programming knowledge you've been taught is required. As a domain engineer, you still have to create the Karel instructions. But you do so for a whole domain of robots, not for a single one!

OBJECTIVE

The students should be able to:

- Understand the generalization from a set of similar architectures to one that applies to any robot in the domain

Robot Software Architecture for Robot Domain



Key:

- Always present
- Sometimes present
- X calls Y in all programs
- X calls Y in some programs

Virginia CENTER of EXCELLENCE for Software Reuse and Technology Transfer

DISCUSSION

This slide shows the solution generation capability that the domain engineer creates for the application engineer. The example is for a robot that picks up litter in a forest, returning when its bag is full or when it has covered the entire forest, whichever occurs first.

Assume the application engineer has made the set of decisions that correspond to this robot. This slide places in context the material contained in Unit 4:

- Domain engineers have figured out the architecture for the robot domain and have implemented a set of reusable components (shown in the buckets at the bottom left corner).
- They have also implemented a "component selector" box which, given a set of decisions (shown as the robot requirements), identifies the needed set of reusable components. These selected components are classes of instructions, rather than specific instructions. In this slide, we select a "Perform Mission" component, which is a component capable of supporting all possible missions a robot might perform. Because the robot zigzags, we don't select a component to negotiate an obstacle.
- Domain engineers have also implemented a "component adaptor" box which, given these decisions and the selected components, adapts the component to the decisions. In this slide, the "Perform Mission" component is adapted to perform the specific mission of picking up litter, returning when the bag is full. Similarly, the "Navigate Terrain" component is adapted to a "Zig-zag."
- The domain engineers have also implemented a "component composer" (the funnel), which takes all the adapted components and combines them into a single solution.

This is what happened when you selected "Generate Solution" in the application engineering environment.

STUDENT INTERACTIONS

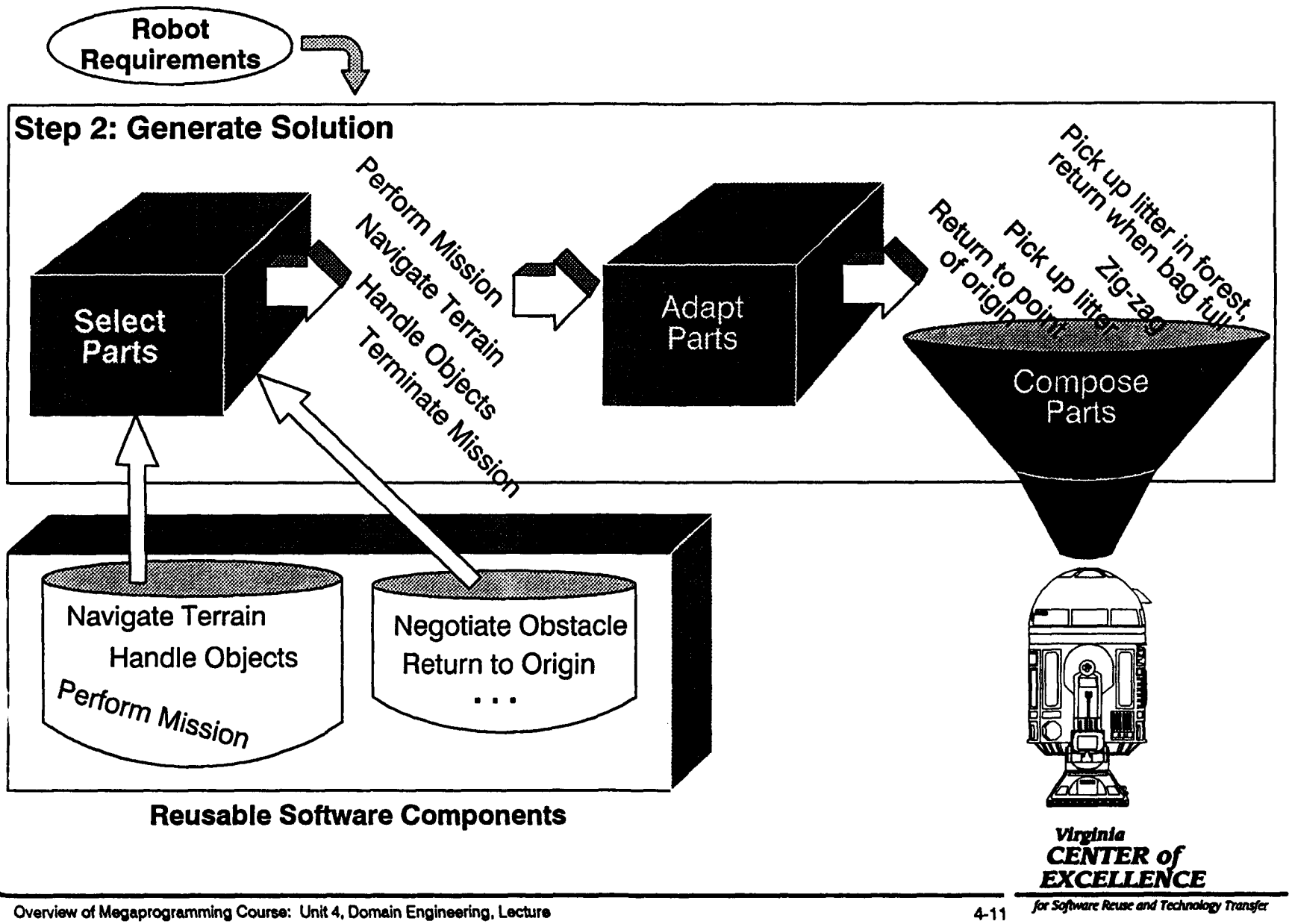
- Where does the robot software architecture (as shown on the previous slide) tie into this picture?

OBJECTIVES

The students should be able to:

- Explain how domain engineers use the knowledge gained from the architecture to help application engineers solve problems
- Define the three major steps in generating a solution

Solution Generation



Virginia CENTER of EXCELLENCE for Software Reuse and Technology Transfer

DISCUSSION

This slide places Unit 4 in context by presenting the picture of megaprogramming. This picture shows the relationship between domain engineering and application engineering.

This unit has discussed everything above the dashed line. Domain engineering results in an understanding of the problems and solutions in a domain. This understanding can be used to develop:

- A process that helps application engineers precisely state a problem.
- A systematic means to generate a solution to a problem. It is based on an understanding of what software components make up a solution (the software architecture) and the procedures for using those components (the generation procedure). Once the domain engineer implements those components, they can be used and reused to solve any problem in the domain.

STUDENT INTERACTIONS

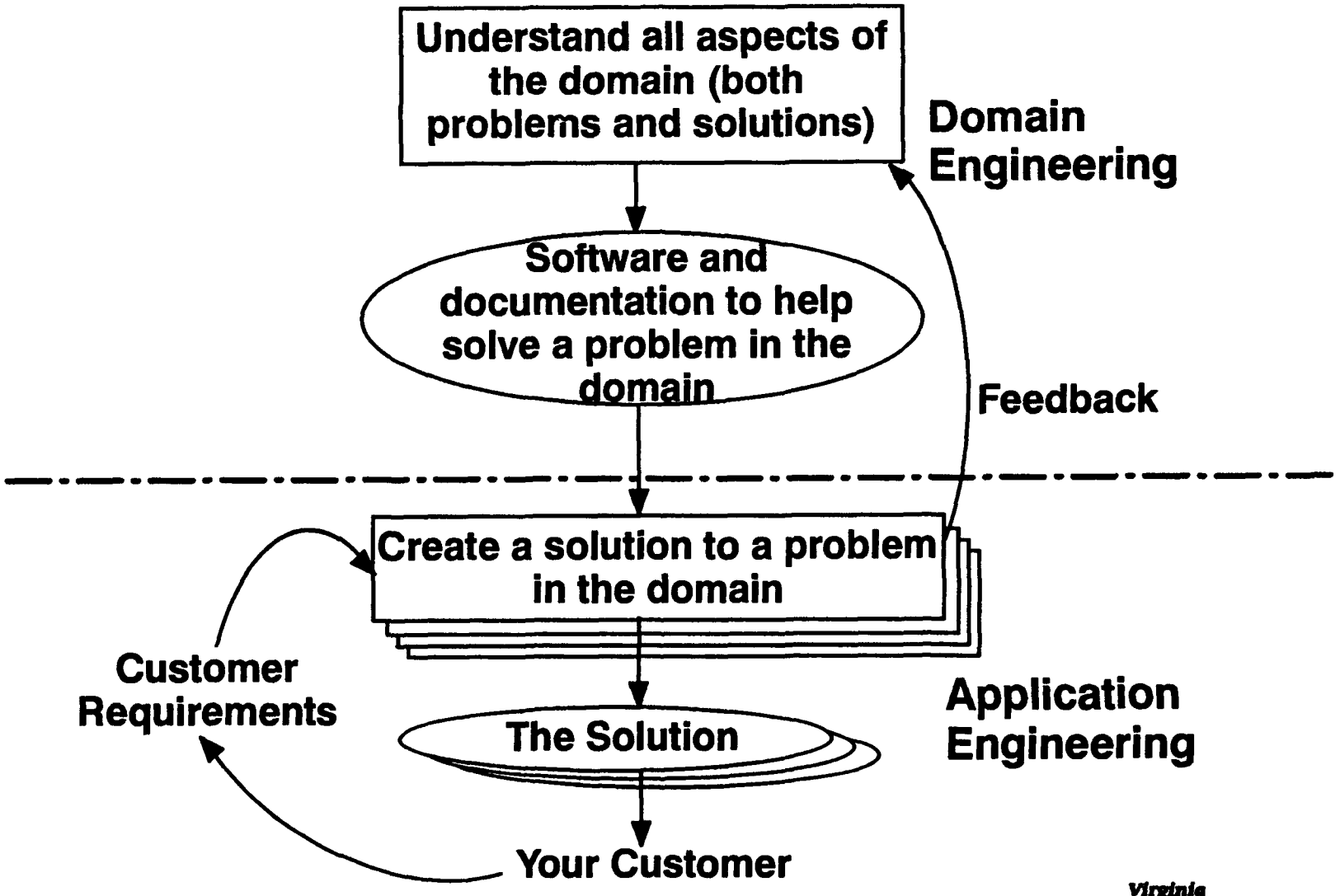
- Now that you know a little more about domain engineering, would you rather be an application engineer or a domain engineer? Why?
- How would you define "megaprogramming"?

OBJECTIVE

The students should be able to:

- Understand the basic functions of domain engineering and application engineering and the relationship between them.

Megaprogramming

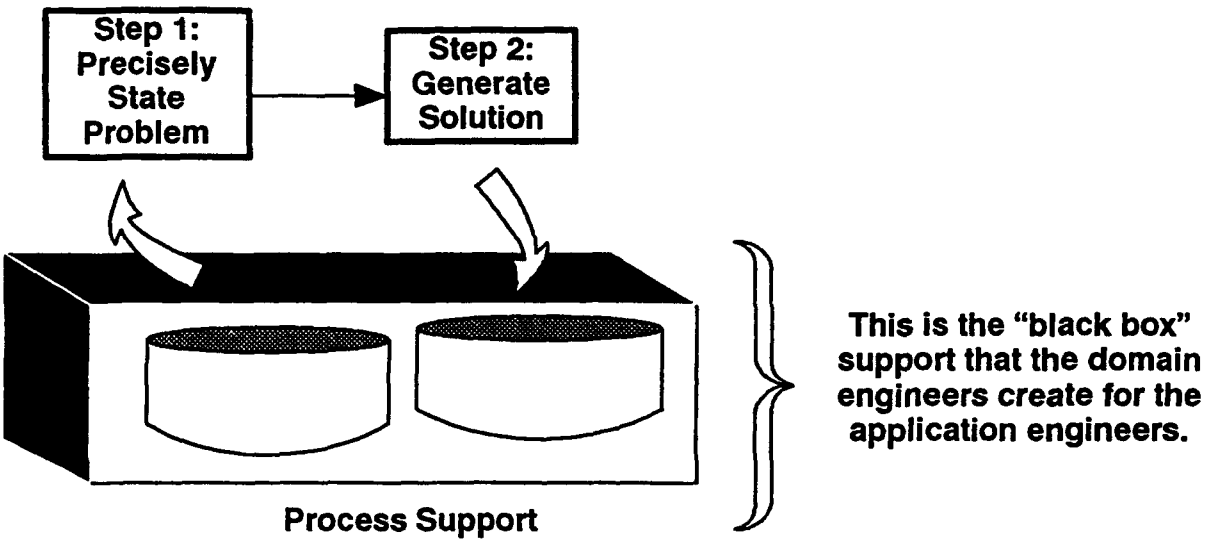


UNIT 4: DOMAIN ENGINEERING

SUMMARY

Domain engineers are responsible for building what the application engineers need to develop solutions. This includes:

- Defining what is in the domain
- Defining the process that the application engineer will follow
- Developing process support (including reusable components) that the application engineer will use to state the problem, validate it, and generate the solution



DEFINING THE DOMAIN

Domain engineers decide what is in a domain.

Application engineers create individual systems. Domain engineers decide the range of systems application engineers can create.

Deciding what is in a domain involves studying the factors that constrain the problems and solutions which form the domain and deciding what problems and solutions are important.

Once the domain engineers know the problems that will be in the domain, they can study them and uncover:

- The commonalities among all problems
- The differences between instances of problems

DEFINING THE PROCESS

An application engineer needs to know what steps to follow in order to develop a solution.

Domain engineers define the process for generating a solution in the domain and develop process support programs to help the application engineer. These support programs include:

- Support for defining and validating the requirements for the solution
- Support for generating the solution

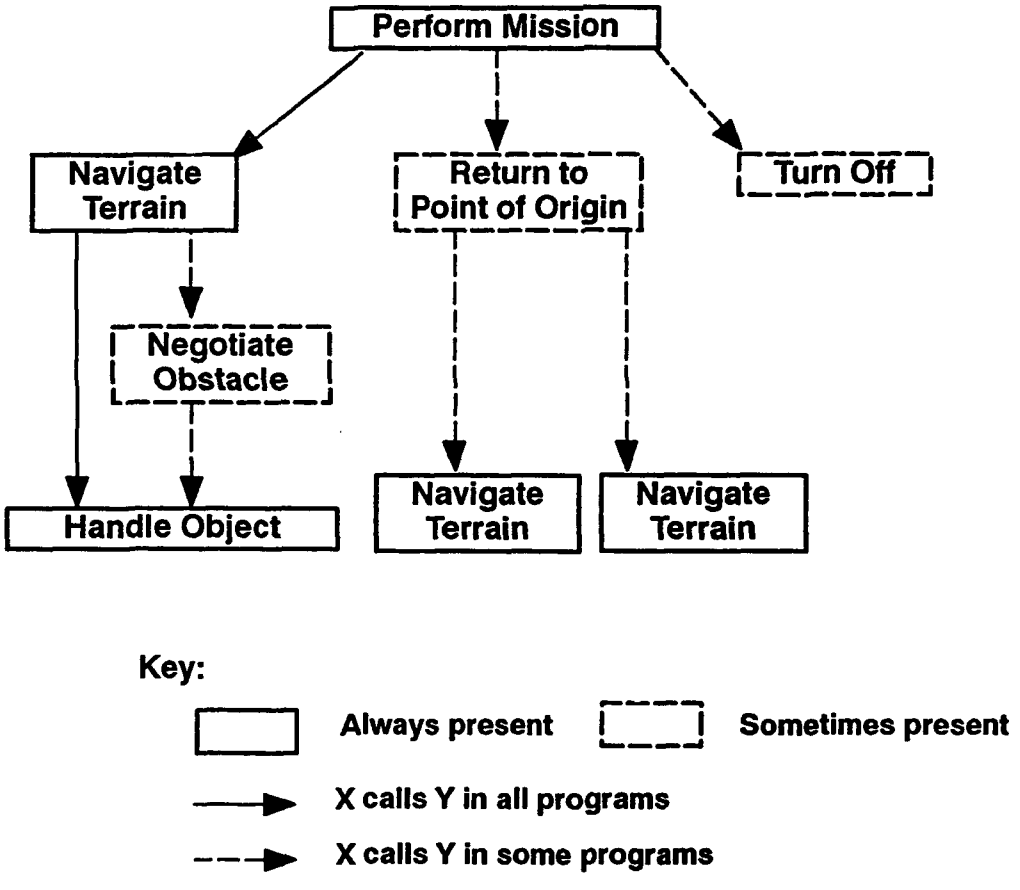
ARCHITECTURES

Every software solution is composed of components (e.g., procedures and functions). Every software solution has an *architecture*, which defines how the components work together.

Domain engineers create a "domain architecture" that:

- Defines the complete set of components used by all solutions in the domain
- Shows what components and interrelationships all solutions have in common
- Shows how individual solutions differ.

The following figure shows the domain architecture for the robot domain.



Domain engineers create these components. They also identify which components are common to all solutions in the domain and which are needed to solve specific problems.

UNIT 4: DOMAIN ENGINEERING

IN-CLASS DISCUSSION

1. Compare results of the laboratory activity. Is there more than one robot software architecture that satisfies the needs of each client? Why or why not?

2. What other kinds of robots could be produced by the URW, domain?

HOMEWORK

1. Considering the domain of the URW, would you, as Chairman of the Board, want to produce robots to:

   a. Plant corn

   b. Pick water lilies

   c. Feed incubator babies

   In making your decision, are there enough similarities to warrant asking your domain engineers to write additional instructions?

2. The instructions in the left column were used to implement the software for a robot that searches a tundra for lost hikers. Each instruction in the left column is an adaptation of an architectural part in the right column. Match each instruction in the left column with the architectural part in the right column.

Instructions	Architectural Parts
1. Advance-north-moving-east-to-avoid-rocks-returning-when-bag-full Move north one unit. If a rock blocks the path, move east around it. If a hiker is found, pick him or her up; if doing so brings the robot to its full capacity, quit this instruction.	A. Perform Mission B. Navigate Terrain C. Negotiate Obstacle D. Handle Object E. Terminate Mission
2. Advance-north-moving-west-to-avoid-rocks-returning-when-bag-full Same as Instruction 1, except that if a rock blocks the path, move west around it.	
3. Sweep-east-returning-when-bag-full Move in a straight eastward line from the current position to the eastern boundary of the area to be searched. If a hiker is found,	

pick him or her up; if doing so brings the robot to its full capacity, quit this instruction.

4. Sweep-west-returning-when-bag-full

   Same as Instruction 3, except move in a straight westward line from the current position to the western boundary of the area to be searched.

5. Sweep-south

   Move in a straight southward line from the current position to the southern boundary of the area to be searched. Ignore any hikers.

6. Sweep-west

   Same as Instruction 5, except move in a straight westward line from the current position to the western boundary of the area to be searched.

7. Pick-up-any-objects

   Pick up as many hikers at the current location as the capacity of the robot allows.

8. Return-when-bag-full

   Search tundra, looking for hikers. When the robot's capacity of hikers has been picked up, or when the entire tundra has been searched, return to the point of origin and turn off.

9. Return-to-starting-point

   From the current position, return to the point of origin.

10. Negotiate-rock-to-east-returning-when-bag-full

    Assumes that there is a rock just ahead of the robot, to the east. Moves the robot such that, when the instruction ends, the robot is just to the east of the rock, at the same latitude as when it started. If any hikers are found while negotiating the rock, they are picked up. If doing so brings the robot to its capacity, the instruction terminates, whether or not the rock has been negotiated.

11. Negotiate-rock-to-west-returning-when-bag-full

    Same as Instruction 10, except assumes that there is a rock just ahead of the robot, to the **west**. Moves the robot such that, when the instruction ends, the robot is just to the **west** of the rock, at the same latitude as when it started.

12. Zig-zag-southwest

    From the current position, zigzag southwest until reaching the southern or western boundary of the area being searched, whichever occurs first. Ignore any hikers.

3. URW, has been approached by the U. S. State Department. The State Department is concerned because it has received reports that embassies around the world have electronic bugs embedded in their walls. The State Department wants to know if URW can supply a robot that can locate these bugs. Fortunately, URW's engineers have just finished developing a new sensor, and they think it can be used for finding bugs. URW therefore decides to modify its robot domain so it can produce this new type of robot in addition to those in its old product line.

    a. For each of the following decisions in the decision-making process, state a requirement for the robot:

        (1) Terrain

        (2) Object type

        (3) Choose if objects are to be carried

        (4) Ending position

        (5) Carrying capacity

    b. Identify the decisions from (1) through (5) whose range of allowed values must be changed to accommodate the new robot.

    (Optional)

    c. Draw the software architecture for the robot, using the domain architecture as a starting point.

    d. Name some instructions from Question 2 that you think could be used without modification.

    e. Name some instructions from Question 2 that could be used with modification. What do you think the modifications might be?

    f. As a domain engineer for URW, what new components, if any, do you think would be necessary?

This page intentionally left blank.

UNIT 4: DOMAIN ENGINEERING

TEACHER NOTES FOR IN-CLASS DISCUSSION

1. Compare results of the laboratory activity. Is there more than one robot software architecture that satisfies the needs of each client? Why or why not?

2. What other kinds of robots could be produced by the URW domain? *This is really an open-ended question and should produce an interesting discussion.*

TEACHER NOTES FOR HOMEWORK

1. Considering the domain of the URW, would you, as Chairman of the Board, want to produce robots to:

   a. Plant corn

   b. Pick water lilies

   c. Feed incubator babies

   In making your decision, are there enough similarities to warrant asking your domain engineers to write additional instructions?

Note to teachers: Familiarity with Karel the Robot *is helpful on the following questions.*

2. The instructions in the left column were used to implement the software for a robot that searches a tundra for lost hikers. Each instruction in the left column is an adaptation of an architectural part in the right column. Match each instruction in the left column with the architectural part in the right column.

Instructions	Architectural Parts
1. Advance-north-moving-east-to-avoid-rocks-returning-when-bag-full Move north one unit. If a rock blocks the path, move east around it. If a hiker is found, pick him or her up; if doing so brings the robot to its full capacity, quit this instruction.	A. Perform Mission B. Navigate Terrain C. Negotiate Obstacle D. Handle Object E. Terminate Mission
2. Advance-north-moving-west-to-avoid-rocks-returning-when-bag-full Same as Instruction 1, except that if a rock blocks the path, move west around it.	

3. Sweep-east-returning-when-bag-full

   Move in a straight eastward line from the current position to the eastern boundary of the area to be searched. If a hiker is found, pick him or her up; if doing so brings the robot to its full capacity, quit this instruction.

4. Sweep-west-returning-when-bag-full

   Same as Instruction 3, except move in a straight westward line from the current position to the western boundary of the area to be searched.

5. Sweep-south

   Move in a straight southward line from the current position to the southern boundary of the area to be searched. Ignore any hikers.

6. Sweep-west

   Same as Instruction 5, except move in a straight westward line from the current position to the western boundary of the area to be searched.

7. Pick-up-any-objects

   Pick up as many hikers at the current location as the capacity of the robot allows.

8. Return-when-bag-full

   Search tundra, looking for hikers. When the robot's capacity of hikers has been picked up, or when the entire tundra has been searched, return to the point of origin and turn off.

9. Return-to-starting-point

   From the current position, return to the point of origin.

10. Negotiate-rock-to-east-returning-when-bag-full

    Assumes that there is a rock just ahead of the robot, to the east. Moves the robot such that, when the instruction ends, the robot is just to the east of the rock, at the same latitude as when it started. If any hikers are found while negotiating the rock, they are picked up. If

doing so brings the robot to its capacity, the instruction terminates, whether or not the rock has been negotiated.

11. Negotiate-rock-to-west-returning-when-bag-full

    Same as Instruction 10, except assumes that there is a rock just ahead of the robot, to the west. Moves the robot such that, when the instruction ends, the robot is just to the west of the rock, at the same latitude as when it started.

12. Zig-zag-southwest

    From the current position, zigzag southwest until reaching the southern or western boundary of the area being searched, whichever occurs first. Ignore any hikers.

Answers: 1–B, 2–B, 3–B, 4–B, 5–B, 6–B, 7–D, 8–A, 9–E, 10–C, 11–C, 12–B

3. URW has been approached by the U. S. State Department. The State Department is concerned because it has received reports that embassies around the world have electronic bugs embedded in their walls. The State Department wants to know if URW can supply a robot that can locate these bugs. Fortunately, URW's engineers have just finished developing a new sensor, and they think it can be used for finding bugs. URW therefore decides to modify its robot domain so it can produce this new type of robot in addition to those in its old product line.

   a. For each of the following decisions in the decision-making process, state a requirement for the robot:

      (1) Terrain

      Answer: The robot is to search buildings.

      (2) Object type

      Answer: The robot is to search for electronic bugs.

      (3) Choose if objects are to be carried

      Answer: The robot is to locate objects, but not carry them.

      (4) Ending position

      Valid answers: The robot is to stop when it locates a bug; the robot is to signal the location of each bug it finds, and continue until it has covered all of the building; or both. That is, URW should consider supplying both types of robots.

(5) Carrying capacity

Answer: The robot will not carry any objects.

b. Identify the decisions from (1) through (5) whose range of allowed values must be changed to accommodate the new robot.

Answer: (1) – new terrain (buildings)
(2) – new object type (bugs)

(Optional)

c. Draw the software architecture for the robot, using the domain architecture as a starting point.

Answer: Draw the architecture with particular nuances based on how the robot terminates its mission.

d. Name some instructions from Question 2 that you think could be used without modification.

e. Name some instructions from Question 2 that could be used with modification. What do you think the modifications might be?

f. As a domain engineer for URW, what new components, if any, do you think would be necessary?

Answer: The old search strategies do not work; however, realizing that is not simple.

Test for *Overview of Megaprogramming* Course

1. In the following table, check whether the task would be done by an application engineer or a domain engineer.

Task	Application Engineer	Domain Engineer
Create the reusable components for a domain.		
Work with the customer to understand the problem.		
Validate the requirements.		
Generate the solution.		
Define what is in the domain.		
Define the process and support needed to generate a solution for a customer.		
Precisely state the problem.		

2. Read the following description of the Car4U Company:

****

Have you ever wanted a car that was taller? wider? bigger? Have you ever shopped the car market and found nothing you wanted (and they still wanted a lot of money for it)? Well, no more, because now there's a new company for the discriminating buyer:

Do We Have a Car 4 U!

The Car4U Company makes cars that are tailored to your every need and desire. You can have car seats that are tailored to your weight, height, and width. You can have bigger windows or smaller windows. You can have bigger trunks or smaller trunks. If you want four-wheel drive, you've got it. If you want your car to be a shade of blue that matches your eyes, we can do that too (in fact, we have over 1000 colors to choose from!). We have engines meant for cruising at high speeds and engines meant for climbing mountains. All in all, Car4U has over 3 dozen options. Each one is meant to help make your car truly your own.

We work with every customer to determine exactly what they want and then develop a car that suits their needs. No longer will you have to wait for the perfect car. Stop by your nearest Car4U store today and see what we can do for you!

****

Based on this description, answer the following questions. Attach separate sheets if needed.

a. What is the output of the Car4U Company's domain engineering activities?

b. What is the output of their application engineering activities?

3. Read the following description of TJ's cash registers domain.

****

Description of TJ's Cash Registers Domain

TJ's Cash Registers domain contains cash registers that can be used in just about any retail situation.

There are several options through which money can be entered into a cash register. The traditional way is to accept cash from the customer and store it in a removable money drawer. In addition to the money drawer, some cash registers are equipped with check imprinting services and/or the ability to scan in credit cards. In all cases, each cash register keeps track of the amount of money that has been received from the customer.

Several retail situations require the use of programmable keys that can store prices for items that are sold frequently. Other price input mechanisms include a price scanning function, a scale for items sold by weight, or the use of the numeric key pad. Only the numeric key pad and the programmable keys are standard, though the number of programmable keys can vary from register to register.

To show prices and to show other information for the cashier and the customer, each cash register has a digital display. Optionally, there may be a separate price display for the customer, either on the back of the register or on a completely separate, smaller display that is above the register and pointed towards the customer. After every transaction, each register automatically outputs a cash register receipt that is printed with the date and time.

Higher-end cash registers can be hooked up to the store's inventory system to either keep track of what the store has in stock (along with a warning message when the stock gets low) or to order items and have the customer pick them up at a separate location.

****

Based on this domain description, answer the following questions. Attach separate sheets if needed.

a. What are the members of the domain?

b. List the similarities between the members of the domain. Be specific.

c. List the differences between the members of the domain. Be specific.

Survey for *Overview of Megaprogramming* Course

Please answer the following questions. The company that developed the course material will use this information to improve the course.

1. Do you feel that you understand the basic principles of megaprogramming after taking this course?

2. Do you see value in megaprogramming?

3. Would you like to learn more?

4. What activity(ies) or example(s) was most helpful to you in understanding megaprogramming?

5. Do you have any other suggestions for how the course can be improved?

This page intentionally left blank.

Test for *Overview of Megaprogramming* Course

Teacher Answers

1. In the following table, check whether the task would be done by an application engineer or a domain engineer.

Task	Application Engineer	Domain Engineer
Create the reusable components for a domain.		X
Work with the customer to understand the problem.	X	
Validate the requirements.	X	
Generate the solution.	X	
Define what is in the domain.		X
Define the process and support needed to generate a solution for a customer.		X
Precisely state the problem.	X	

2. Read the following description of the Car4U Company:

****

Have you ever wanted a car that was taller? wider? bigger? Have you ever shopped the car market and found nothing you wanted (and they still wanted a lot of money for it)? Well, no more, because now there's a new company for the discriminating buyer:

Do We Have a Car 4 U!

The Car4U Company makes cars that are tailored to your every need and desire. You can have car seats that are tailored to your weight, height, and width. You can have bigger windows or smaller windows. You can have bigger trunks or smaller trunks. If you want four-wheel drive, you've got it. If you want your car to be a shade of blue that matches your eyes, we can do that too (in fact, we have over 1000 colors to choose from!). We have engines meant for cruising at high speeds and engines meant for climbing mountains. All in all, Car4U has over 3 dozen options. Each one is meant to help make your car truly your own.

We work with every customer to determine exactly what they want and then develop a car that suits their needs. No longer will you have to wait for the perfect car. Stop by your nearest Car4U store today and see what we can do for you!

****

Based on this description, answer the following questions. Attach separate sheets if needed.

a. What is the output of the Car4U Company's domain engineering activities?

Domain engineering would (1) create all of the different car components that would be needed to make a car, (2) create the ordered list of questions that the car salesperson would ask the customer, and (3) create the instructions for how the actual car builders would put together the car based on the specific needs of a specific customer.

b. What is the output of their application engineering activities?

Application engineering would (1) talk with the customer to understand what the customer wanted in a car, (2) use that understanding to come up with a precise statement of what was needed in the car, (3) make sure that this precise statement was what the customer wanted, and (4) generate the car (with help from the actual car builders) that met the customer's specific need.

3. Read the following description of TJ's cash registers domain.

****

Description of TJ's Cash Registers Domain

TJ Inc. makes cash registers that can be used in just about any retail situation.

There are several options through which money can be entered into a cash register. The traditional way is to accept cash from the customer and store it in a removable money drawer. In addition to the money drawer, some cash registers are equipped with check imprinting services and/or the ability to scan in credit cards. In all cases, each cash register keeps track of the amount of money that has been received from the customer.

Several retail situations require the use of programmable keys that can store prices for items that are sold frequently. Other price input mechanisms include a price scanning function, a scale for items sold by weight, or the use of the numeric key pad. Only the numeric key pad and the programmable keys are standard, though the number of programmable keys can vary from register to register.

To show prices and to show other information for the cashier and the customer, each cash register has a digital display. Optionally, there may be a separate price display for the customer, either on the back of the register or on a completely separate, smaller display that is above the register and pointed towards the customer. After every transaction, each register automatically outputs a cash register receipt that is printed with the date and time.

Higher-end cash registers can be hooked up to the store's inventory system to either keep track of what the store has in stock (along with a warning message when the stock gets low) or to order items and have the customer pick them up at a separate location.

****

Based on this domain description, answer the following questions. Attach separate sheets if needed.

a. What are the members of the domain?

The members of TJ's Cash Registers domain are cash registers that could be built by TJ Inc.

b. List the similarities between the members of the domain. Be specific.

(1) Removable money drawer
(2) Ability to keep track of the amount of money received by customers
(3) Numeric key pad
(4) Existence of programmable keys
(5) Digital display
(6) Ability to output a cash register receipt

c. List the differences between the members of the domain. Be specific.

(1) Check imprinting services
(2) Ability to scan in credit cards
(3) Price scanning function
(4) Scale for items sold by weight
(5) Number of programmable keys
(6) Price display on back of register
(7) Separate price display pointed towards the customer
(8) Hook-up to store's inventory system to keep track of what's in stock
(9) Hook-up to store's inventory system to order items to be picked up at separate location

****

Survey for *Overview of Megaprogramming* Course

Teacher Answers

There are no right or wrong answers on this section. A suggestion for this survey would be to hand it to the students after they have completed the test and give them extra credit if they fill it out and hand it in the next day.

This page intentionally left blank.

List of Abbreviations and Acronymns

4GL	fourth-generation language
SGA	Student Government Association
3GL	third-generation language
URW	United Robot Workers, Inc.

This page intentionally left blank.